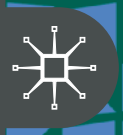


palgrave pivot

Applied Quantitative Finance

Using Python for Financial Analysis

Mauricio Garita

palgrave
macmillan

Applied Quantitative Finance

Mauricio Garita

Applied Quantitative Finance

Using Python for Financial Analysis

palgrave
macmillan

Mauricio Garita
Universidad Francisco Marroquín
Guatemala City, Guatemala

ISBN 978-3-030-29140-2 ISBN 978-3-030-29141-9 (eBook)
https://doi.org/10.1007/978-3-030-29141-9

Cover illustration: © Melisa Hasan

This Palgrave Pivot imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

I want to thank God, my wife Sonia, Míkel and Maia for being the reason for keeping moving forward.

Introduction

This book is aimed for students, professionals, academics and everyone who wants to learn Python and its application to the stock market. Therefore, the book begins with the simple implementation of Python, advances to statistical methods and ultimately reaches the creation of portfolios.

The book also provides a clear understanding of the different discussions that surge between academic and professional world in the area of finance. As the reader will see, there are references throughout the book so that the learning experience may continue after the book is finished.

When considering the different aspects of programming the author centered on the writing of the code as a whole, so that it can be followed by the reader. This allows a better learning and an implementation of the process by the reader, giving a better knowledge about the process, the errors and how to implement solutions.

As a final note, this book does not aim to be a in-depth finance book or programming book. This book is written from a practitioner point of view from finance using programming. The reader should see and analyze this book from a practitioner perspective, with the purpose of learning the application of different statistical, mathematical and financial concepts in the growing language that is Python.

As an author, this book is an effort to close an important gap for those interested in financial programming and I hope that it opens the doors to new possibilities.

What This Book Is and Is Not

This book is not a theoretical book that will explain each and every detail of each indicator presented. This book centers in applying finance to the different indicators to offer a hands-on experience.

This book does not cover all the aspects of finance. For example, the book is centered in technical, quantitative and risk analysis of the stock market, but it does not cover each and every avenue. This book does not contain options and futures, Montecarlo Simulations and binomial trees. The reason is that this book aims to be an introductory to intermediate level. Considering advanced level there are books far more detailed on this aspects.

This book does not aim to explain programming language to the reader. It explains the easiest way to program a portfolio, a MACD, a VaR and other financial instruments. Also, the code is simple and clean so that the reader is not overwhelmed by programming. I truly believe that we can learn to program if we start from the basics and this book aims for this.

This book does not aim to be perfect. Its aim is to open a discussion considering finance and the growth of the different methods of the internet. Therefore, the book is explained through Yahoo API because it is the present live information at every second. Even though, everything can be elaborated in Excel after uploading the document to Jupyter notebooks or Google Collaboratory.

Finally, during the writing of this book, I began reading different scripts from different authors trying to find the easiest way possible to understand Python. This book is an effort to make things simple.

Thank you for believing in this book. Thank you for giving it the opportunity of being in your hands, in your reading, supporting your future. This book is a tool that I wish I had at the beginning of my career. Simple enough to understand but solid enough to be reliable.

Sincerely, Mauricio Garita

Contents

About the Author

Mauricio Garita began his studies at the Universidad Rafael Landivar in Guatemala studying Economics and Business Administration. Once he graduated, he went to Manchester Business School to study a master's in international business and to el Instituto de Estudios Bursátiles (IEB) to study a master's in asset management. He received his Ph.D. from the Pontificia Universidad de Salamanca focusing in Sociology and Politics writing his thesis on the economic impact of the Civil War in Guatemala.

He has worked with the office of the World Bank in Guatemala focusing on financial and economic aspects, he directed the Department of Business Intelligence in the Secretariat for Central American Economic Integration (SIECA) and the Department of Business Intelligence of the Private Sector (CACIF) at Guatemala. He also created the Department of Academic Relations at the Central American Institute of Fiscal Studies (ICEFI).

In academics, Mauricio has taught at a masters and bachelors level courses focused on economics and finance at the Universidad Rafael Landivar, Universidad del Valle, Universidad del Istmo and lately at Universidad Francisco Marroquin. He was part of the founders of the master's in advanced finance at the Universidad del Valle in Guatemala and researcher and professor in finance at Universidad Francisco Marroquin.

On a business level, he founded Simpleconomics which centers on personal and business finance combined with philosophical views from the stoic philosophy. He is one of the founders of the Stoic Chapter

in Guatemala. He also created a *podcast* with the name *Simpleconomics* which is a podcast focused on finance and philosophy.

His research can be read in Business and Politics, Emerging Economies and Multinational journals to name a few, in the Routledge book written in collaboration with John Spillan and Nichols Virzi, in the Rethinking Taxation in Latin America book by Palgrave Macmillan and in newspapers such as Prensa Libre of Guatemala and the economic magazines such as El Economista and Estrategia y Negocios for Central America.

List of Figures

Why Python?

Using Stock Market Data in Python

Statistical Methods Using Python for Analyzing Stocks

Elements for Technical Analysis Using Python

Valuation and Risk Models with Stocks

Value at Risk

List of Tables

List of Equations

Statistical Methods Using Python for Analyzing Stocks

Elements for Technical Analysis Using Python

Valuation and Risk Models with Stocks

Value at Risk

Why Python?

Abstract This chapter focuses on the importance of Python for the financial world and its implications in the present and the future of the financial industry. With the growth in the easiness for accessing information, Python becomes relevant in creating faster models for analyzing data. The second objective of this chapter is to understand how to install Python, analyzing other aspects for analyzing data with Google Collaboratory without the need of having Python installed in the computer, knowing how to install Anaconda and using Jupyter Notebooks.

Keyword Importance of Python · Jupyter Notebooks · Anaconda · Google Collaboratory

When Guido Van Rossum created the languages, he was aiming for a better, easier language to handle Amoeba which is a microkernel-based operating system. The creating began in 1989 and in 1991 it was posted by the author to USENET (Python Organization 2020).

From there, Python has grown into one of the most used programming languages in the world. O'Reilly (2020) has mentioned that in their online learning environment, Python is preeminent and that it accounts approximately 10% of the usage in their website. The growth of the language is demonstrated in the new application programming

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_1

interface (API) created by different companies in the field of finance such as Quantopian, Quandle, Yahoo Finance and Bloomberg to mention a few.

As mentioned by Bloomberg (2019) the "*the lingua franca of finance is shifting from Microsoft Excel to the open-source programming language Python*". The use of Python has gone from a desire to a need when engaging into the world of finance and programming. It has become a necessity for finance professional despite the background.

The Future Fund—USD140 billion,[1] created in 2006 in Australia, is teaching its investment team Python in order to analyze that more efficiently and rapidly (Burgess and Wells 2020). It has become a must in the industry and will continue to grow in further years.

Another important aspect considering the growth of Python is the easiness of its use. The program has a simple and easy to learn syntax, it is versatile, it is now widely used and has a community that supports the process of knowledge and upgrading the code (Shaik 2018). Considering these aspects, the usual question that arises is when choosing between R and Python.

R was developed by Ross Ihaka and Robert Gentleman in 1992 and since then it has grown into a widely used language, specifically in the statistical methods area. The comparison between both languages is based on the popularity and its use. In 2017, Stack Overflow presented a survey in which they established that 45% of data scientists use Python in comparison with just 11.2% using R (Kan 2018). The final decision is based on the user, but the usual reason is that Python is easy to use and with the use of Jupyter Notebooks[2] it has become extremely helpful considering data analysis (Fig. 1).

As a conclusion on why use Python for finance, the answer can be divided into the following categories:

– *Ease of use*: The language is extremely easy to use and to understand. The process of accessing data from an Excel or an API is simple. The different commands are intuitive and the website www.python.org offers an excellent resource for understanding the process in the command module.

[1] As of February 9th 2020.

[2] For more information concerning Jupyter Notebooks visit next chapter.

CRITERION	TYPE	RATING	EXPLANATION
Origin/Focus	Python	●●●●●	General programming language. Origin: Computer Science.
	R	●●●●●	Specialist statistics software. Origin: Statistics.
Number and quality of procedures/ Modelling	Python	●●●●●	Higher number in absolute terms. Very large number of procedures in the field of ML and modelling.
	R	●●●●●	Highest number of procedures/packages in the field of ML and modelling.
Data preparation	Python	●●●●●	Slight advantages in (the field of) automation/ processes. Equal in terms of general data preparation.
	R	●●●●	Equal in terms of general data preparation.
Visualization	Python	●●●●	Good visualization options via packages. R better in the past, Python is catching up. "Matter of taste".
	R	●●●●●	Good visualization options via packages. R better in the past, Python is catching up. "Matter of taste".
Performance	Python	●●●●●	Depending on task. Slightly better performance on average.
	R	●●●●	Depending on task. Slightly poorer performance on average.
Learnability	Python	●●●●●	Slightly easier to learn. Flatter learning curve.
	R	●●●●	Steeper learning curve.
Community / Web support	Python	●●●●●	Large community. Many online sources for help and coding. Community more programming-affine.
	R	●●●●	Large community. Many online sources for help and coding. Community more statistics-affine.
Connectablity of ML and AI libraries / API	Python	●●●●●	Better support (documentation) of latest Deep Learning and Big Data libraries (e.g. Tensorflow, Spark).
	R	●●●	Less support (documentation) of latest Deep Learning and Big Data libraries (exception: H2O.ai).
Prevalence / Trend	Python	●●●●●	Widely used. Most used language in the field of Data Science.
	R	●●●●	Widely used. Used to be most used language for a long time. Strong trend: Python is overtaking R.
Available know-how	Python	●●●●	For IT developers there is a stronger focus on Python at universities, although R is also taught. Currently, university graduates are more likely to enter the job market with Python know-how.
	R	●●●●	Statistics focus on R at university, although openness for Python is given. Employees with 5-10 year working experience were taught R at university.

Fig. 1 Comparison between R and Python (*Source* [Pfeiffer 2019])

- *Jupyter Notebooks*: In the following chapter the use of Jupyter Notebooks will be discussed deeply, in the meanwhile it is important to state that the use of notebooks is helpful in the programming process as well as when elaborating different plots.
- *Plots*: In the book the Matplotlib library will be used when elaborating plots. Matplotlib is a very useful library for elaborating 2D plots. The plots are easy to configurate and share.

The main reason for using Python is that it is growing faster than predicted in the finance sector, giving way to a faster and better analysis of the stock market. The use of API for gathering information without the hassle of downloading an excel, getting to know where the information should be kept or knowing which folder the right folder is to use. The future of finance is centered on the capacity of the new technology and Python, for the author, is the best vehicle.

Installing Python in the Computer

The following is based on the installation of Python with the Anaconda Distribution. For the installation of Python in its original form the user can visit www.python.org/downloads/ and choose from the different options of Operating System. For the purpose of the book the following instructions are for the installation of Python using Anaconda Distribution.

The process for installing Python is practically the same for macOS users as for Windows users. Given the scope of the present book, these are the two operating systems we will be addressing in the installation of Python. Once installed, the interface of Python does not change concerning the operating system.

The first step is to download Anaconda Navigator. As stated before, the reason for using Anaconda Navigators is that it simplifies greatly the process of installing packages and environments needed for Python. The interface is also useful for other programming languages such as R. It also provides the possibility of creating uploading information into their own Cloud with the possibility of sharing it to others.

Anaconda Navigator can be downloaded from their main webpage https://anaconda.org/ by selecting ≪Download Anaconda≫ from the menu. This will automatically read the interface of the computer and guide you to selecting it for macOS, Windows or Linux.

For this book, it is recommended that the version of Python that should be downloaded is *Python 3.8 version*. This version has important changes in comparison with its last version, the 2.7, mostly in the writing of the code. Therefore, it could create confusion on the user given that certain commands and lines of code will be written differently.

The version that is recommended is the 3.7 version which is the one used in the book. There are certain things that are done differently in the 2.7 version than they are done in the present book, so if the apprentice is working with a 2.7 version, he must keep this in mind.

When installing the package, the version that *downloads* by pressing in the 3.6 version will suffice. There are two other versions as the 64-Bit Graphical Installer and the 64-Bit Command line installer but for the purpose of the present book the first one will suffice. Once the package is downloaded the process is as follows (for Mac OS):

Step 1: The software has to recognize if it is capable of running the software.
Step 2: Select a destination that is appropriate. The recommendation is that it is installed in a specific disk.
Step 3: Perform the Standard Installation taking into consideration the space required. Approximately 2.13 GB.
The installation should be easy and without a problem. If a problem occurs the Support Community or the support offered by Anaconda´s team should be of help. The support page can be visited at the following link: https://www.anaconda.com/support/.

Once the Anaconda Navigator is installed, on the left side a menu can be read with the categories Home, Environments, Projects (beta), Learning and Community. For installing a package, it is important to close the Jupyter Notebook that has been worked on, because if not that package will not be read by the Notebook.

Once downloaded an installing screen just as the one below will guide you on the installations.

When installing Anaconda in the *Advanced installation options* it is recommended to set Python 3.8 as default. This will allow the possibility of sharing information with other programs which is useful for the rest of the book.

The installation should be followed according to recommendation and by attending this recommendation you will have the Anaconda Navigator installed in your computer.

When the user opens the Anaconda Navigator this will lead to the main screen. In the main screen the user will see on the left side a menu with the items Home, Environments, Projects (beta), Learning and Community. More detail on each of the items in the menu will be given as the book advances.

On the center the user will have a different option such as Jupyterlab, Jupyter Notebook, qtconsole, spyder, glueviz, orange3, rstudio, vscode. The book will be using mostly the Jupyter Notebook environment for the exercises. Finally, on the right-hand side the user will see ≪Sign in into Anaconda Cloud≫ which gives the possibility of uploading the notebooks, something that we will explore at the end of the book.

The second step, after installing Anaconda Navigator is to launch and accommodate our Jupyter Notebooks. By selecting to launch Jupyter Notebooks, a webpage of the user default search engine will be opened and on the search bar an address such as localhost:8888/tree will appear. This is the user's main directory. This is important because:

- In the directory the user Jupyter Notebooks will be stored for accessing them.
- Any information such as Excel documents or images will be retrieved from this directory.
- It will be very useful for uploading Jupyter Notebooks to the Cloud.

Using Jupyter Notebooks with Python

The easiness mentioned in the earlier chapter can be made into more accessible with the use of Jupyter Notebooks. The Jupyter Project began in 2014 as an open-source software that works perfectly with Python.

Jupyter Notebooks are recommended for data science and in this case for financial analysis. The main reason for using the Jupyter Notebook is for the communication between Python and the computer. The conjunction of both leads to a better approach when coding, visualizing data, creating kernels and installing updates as well as libraries (365 Data Science 2020). For installing Jupyter Notebooks in the computer there are two procedures:

– Installing Jupyter Software directly: In www.jupyter.org/install there is a simple process to follow the installation for the use of Jupyter on the user's computer. In this website it is possible to install in Windows, Mac and Linux.

– Installing Anaconda: This is the procedure that is recommended if the user hasn't used the command prompt in Windows or its similar in Mac and Linux. Anaconda can be installed by visiting the webpage www.anaconda.com/distribution where it can be installed in Windows, Mac and Linux.

The reason why Anaconda is the recommended procedure is because it is the easiest way to install packages, manage libraries, analyze data and visualize results (Anaconda 2020). For the users that are not proficient with the command prompt or similar in Linux or MacOS, it is recommended to begin with the Anaconda Distribution.

For the present book, all the code has been written in Jupyter Notebooks using Anaconda Distribution. The installation of the packages and the use of the plots with Matplotlib has been simplified by the use of this distribution system.

Understanding Jupyter Notebooks

The growth of Jupyter Notebooks in the data science field has been based on the capacity of writing code that can be easily shared and has a comprehensible order for others to understand and track any changes. Since its creation in 2014, the multi-language interactive web application, is the most useful tool when elaborating financial code (Das 2020) (Fig. 2).

After installing Anaconda Navigator is to launch and accommodate our Jupyter Notebooks. By selecting to launch Jupyter Notebooks, a webpage of your default search engine will be opened and on the search bar an address such as localhost:8888/tree will appear. This is your main directory. This is important because:

- In the directory your Jupyter Notebooks will be stored for accessing them.
- Any information such as Excel documents or images will be retrieved from this directory.
- It will be very useful for uploading Jupyter Notebooks to the Cloud.

Once Jupyter Notebooks is open the program leads to the main directory. This is the directory where Python is installed. In the next segment, we will discuss how to change the Python directory.

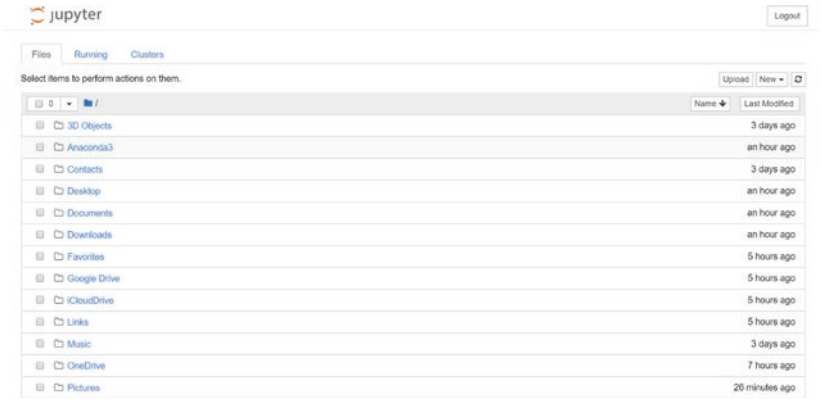


Fig. 2 Jupyter Notebooks (*Source* Obtained from the computer of the author)

On the top left-hand side there are three important aspects for Python:

- Files
  - Here are the new and the old files. You can create a new file; you can move files into a new folder or create a new folder. Also, you can access files that have been created previously.
- Running
  - This will give you the information concerning which programs are being used.
- Clusters
  - This will not be a topic for the book since it involves sharing information with other data science webpages.

On the left-hand side there will be two options:

- Upload
  - This will be used to upload a different document, excel or Jupyter Notebook to the directory.
- New
  - Depending on the version and the system, by selecting new you can build a new Python Notebook, a text file, a folder or prompt the Terminal.

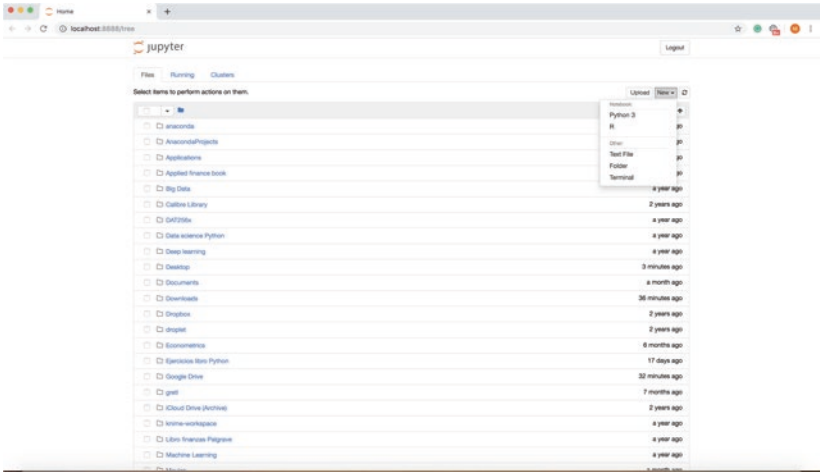


Fig. 3 Jupyter Notebook—selecting Python (*Source* Obtained from the computer of the author)

Once Jupyter Notebook is launched the first step is to create a *folder* for the work to be archived. This can be done by selecting the *new* bar on the left and then from the options selecting folder (Fig. 3).

Once the folder has been created, it can be renamed by choosing the folder and then choosing *Rename* that will appear once you choose the folder that will be modified (Fig. 4).

Once the folder is created then the python files can be added by choosing the top left sorting menu and choosing python.

Some important aspects before starting in Jupyter Notebooks (Fig. 5):

- *Finding your directory*: Type in the bar *pwd* and press enter. This will tell where the file is so that it is retrievable, and it can be used later in the process.
- *About the Kernel*: A *kernel* is basically the program that is composed by the *script* (what you write on the notebook). It is important to restart, shutdown or reconnect the *kernel* depending on the necessity. When running the program, always restart the *kernel* if you have been logged out of your computer or been away from the computer for some time.

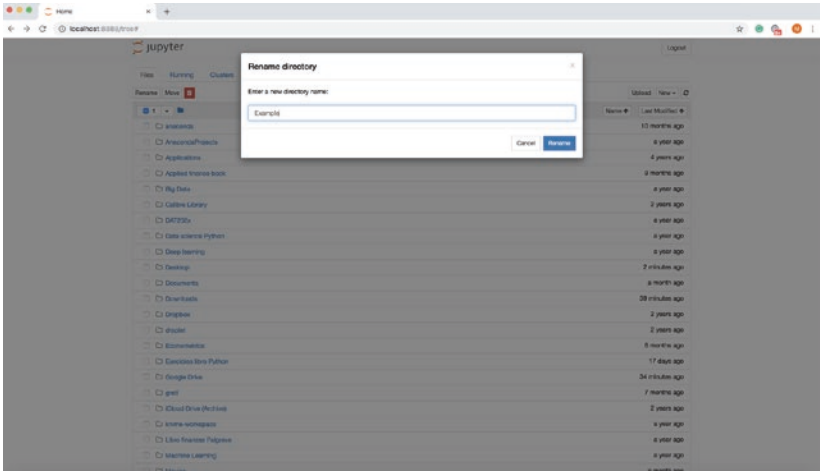


Fig. 4 Creating a folder (*Source* Obtained from the computer of the author)

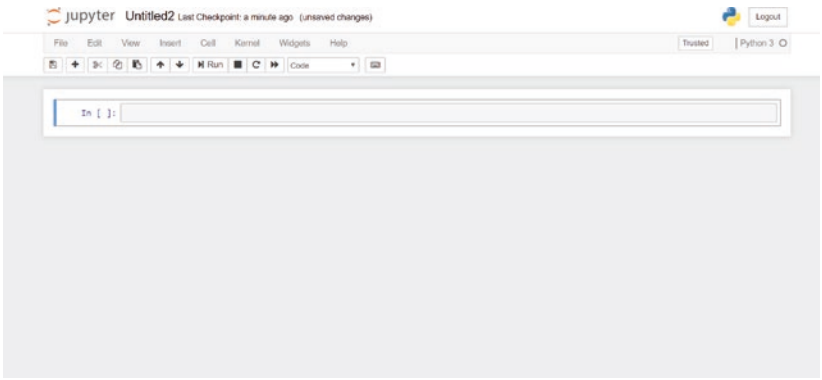


Fig. 5 A sample of Jupyter Notebooks (*Source* Obtained from the computer of the author)

By selecting new and pressing in the Python 3 items, a new window will be displayed. This is a Jupyter Notebook where all the commands and information will be implemented. To understand how a Jupyter Notebook works it is important to define each of the aspects:

- The name of the notebook
  - You will find it on the upper side of the window. When it is new, the name is usually "Untitled". By pressing on Untitled you can change the name to a name more familiar to what you will be working on.
- The ribbon
  - On the ribbon you will find different options. The first one is the File option which gives the possibility of opening documents, making a copy, renaming, saving and checkpoint, revert, print preview, download and Close and Halt. We will see in detail in the book each of the steps as we continue working.
  - The second option will be Edit. It is similar to other programs such as the Office package where you can change the different aspects of documents.
  - The other options are Insert, Cell, Kernel, Widget and Help. We will see in detail these aspects since they are relevant to the usage of Python.

On the center of the Jupyter Notebook you will find the command prompt line with a In [ ] and a gray line. Here is where the code is written and then executed. Once executed you will have an Out [ ] line of code with the result.

Finally, Jupyter Notebooks are very user-friendly and if a mistake is made, it leads to a description of the error which allows to know a mistake in an easier way and this allows it to be solved.

Installing a package (Fig. 6):

When selecting Environments there are two columns of information. The first column is titled Search Environments which leads to the different environments that will be used in the process. For example, there are different environments depending on the use of the environment is deep learning or Big Data. On the right of the environment there is the package section which can be sorted by the installed/not installed on the upper left. If one chose to see the installed packages it will display all the packages that have been installed in the computer.

On the top right there is a *search packages* bar that is very useful when searching for a specific package such as NumPy, a package that is used consistently throughout the book (Fig. 7).

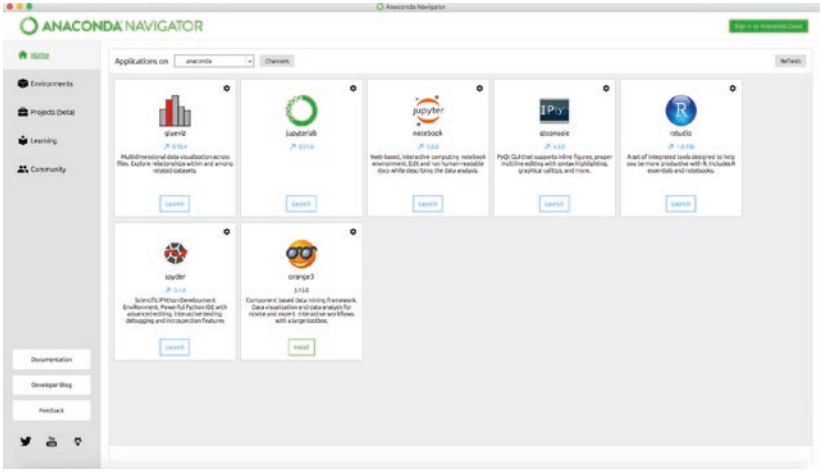


Fig. 6 Installing a package (*Source* Obtained from the computer of the author)

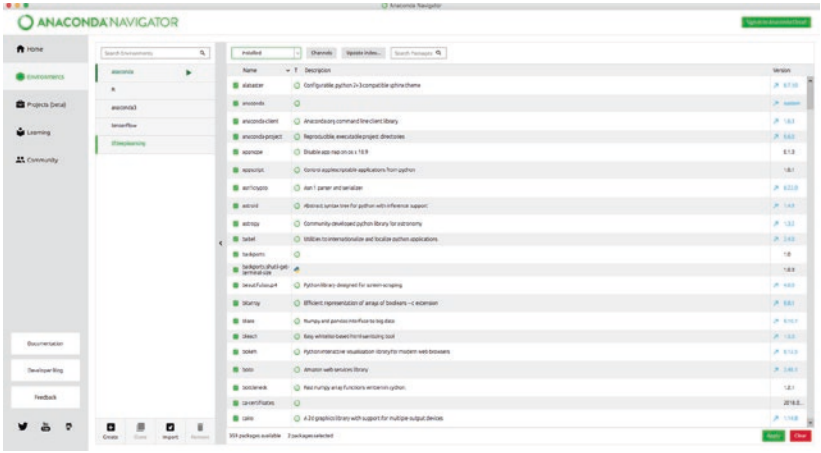


Fig. 7 Selecting package to install (*Source* Obtained from the computer of the author)

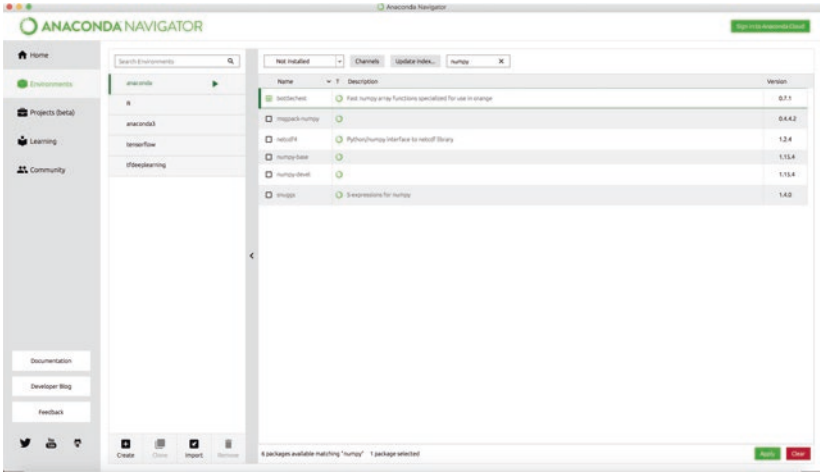


Fig. 8 Searching for a package (*Source* Obtained from the computer of the author)

To install a package is simple using Anaconda Navigator. It can be more difficult by using the Terminal, this is one of the most important reasons for using the Anaconda Navigator software. If a package is not installed there are two choices for searching the package (Fig. 8):

1. Sorting by not install
2. Using the search bar for the specific package.

Once the package is located, the small square box should be selected for download. On the bottom right of the page is an *apply* or *clear* button, select the apply button for the installation of the package.

By choosing to apply the package a small window will appear that will search for the package, modify it and solve any problem. Be aware that this may take some time depending if the package is download and if there is any information that is needed to download the package. Once the package is installed it will appear in the installed section and it is available to be used when installing a library (see next chapter) (Fig. 9).

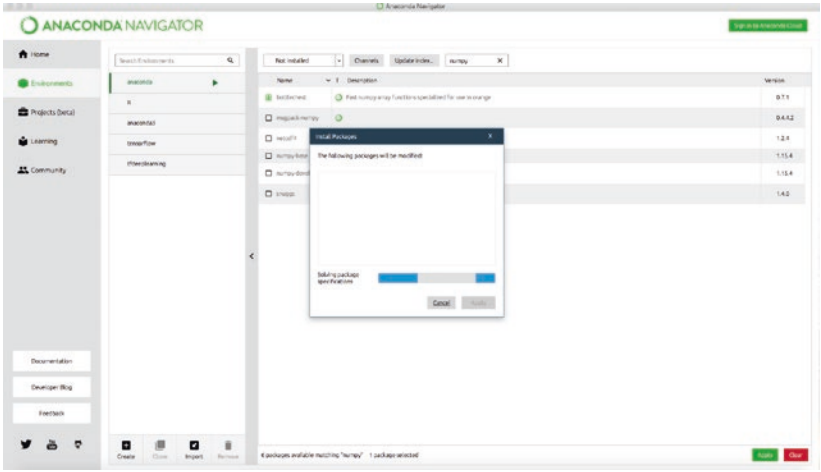


Fig. 9 Process of installing a package (*Source* Obtained from the computer of the author)

Once the packages are installed, the launch Jupyter Notebook can be launched by going back to *Home* and pressing on launch under the Jupyter Notebook symbol. This will open a search engine window and is the first process to create a notebook.

Using Google Colab

Google Colab or Google Colaboratory is a new interface to execute Python code created by Google. One of the most important aspects of using Google Colab is that the notebooks can be shared easily with others to collaborate (therefore the name) in the programming project. The platform is *free* and it can be accessed by using the *Gmail* account.

To create a Colab notebook the following process should be followed (Fig. 10):

Step 1: Create a Drive in https://drive.google.com/.
Step 2: Once the Drive is created, the new tab can be accessed.

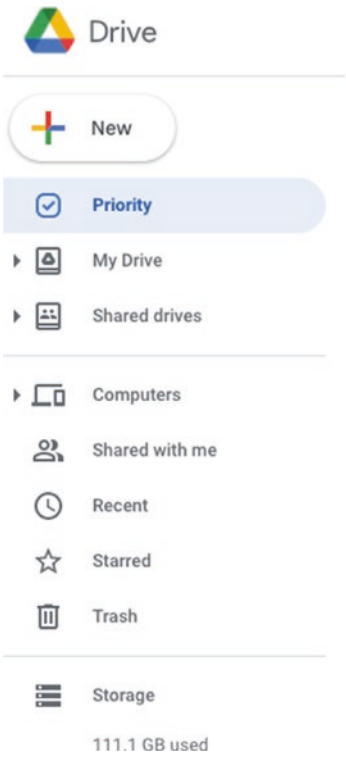


Fig. 10 Creating a new Colab document

When pressing the *New* tab to create a new document using the Google option, one should look for the Google Colaboratory. It is usually in the *more* option (Fig. 11).

Once the Google Colaboratory is found, then it can be selected.

Step 3: Name your notebook by pressing beside the *ipynb* located on the top right. For this example the name will be changed to *example* (Fig. 12).

Once the document is created, the symbol of *Google Colaboratory* in the top left can be pressed to return to all the documents that have been created (Fig. 13).

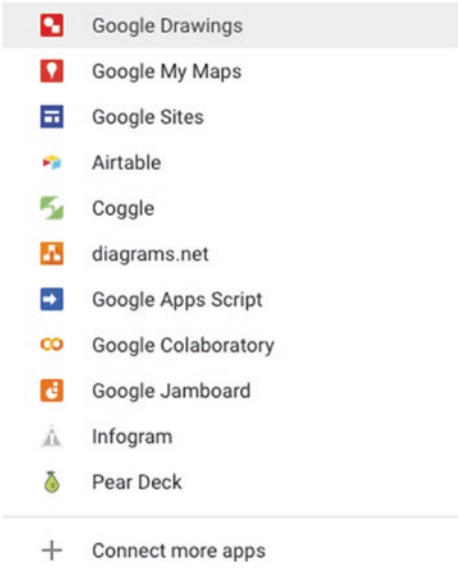


Fig. 11 Google Colaboratory

Fig. 12 Changing name to notebook in Google Colab

Fig. 13 Accessing Google Colab documents

References

365 Data Science. 2020. *Why Python for data science and why Jupyter to code in Python Articles 11 min read*. n.d. Accessed March 2, 2020. https://365datascience.com/why-python-for-data-science-and-why-jupyter-to-code-in-python/.

Anaconda. 2020. *Anaconda distribution*. Accessed March 2, 2020. https://www.anaconda.com/distribution/.

Bloomberg Corporation. 2019. *Bloomberg puts the power of Python in hedgers' hands*. 8 March. Accessed March 2, 2020. https://www.bloomberg.com/professional/blog/bloomberg-puts-power-python-hedgers-hands/..

Burgess, Matthew, and Sarah Wells. 2020. *Giant wealth fund seeks managers who can beat frothy market*. 9 February. Accessed March 2, 2020. https://finance.yahoo.com/news/giant-wealth-fund-seeks-managers-230000386.html.

Das, Sejuti. 2020. *Analytics India Magazine*. 27 February. Accessed March 17, 2020. https://analyticsindiamag.com/why-jupyter-notebooks-are-so-popular-among-data-scientists/.

Kan, Chi Nok. 2018. *Data Science 101: Is Python better than R?* 1 August. Accessed March 2, 2020. https://towardsdatascience.com/data-science-101-is-python-better-than-r-b8f258f57b0f.

O'Reilly. 2020. *5 key areas for tech leaders to watch in 2020*. 18 February. Accessed March 2, 2020. https://www.oreilly.com/radar/oreilly-2020-platform-analysis/.

Pfeiffer, Frank. 2019. *R versus Python: Which programming language is better for data science projects in Finance?* 28 May. Accessed March 2, 2020. https://finance-blog.arvato.com/r-versus-python-in-finance/.

Python Organization. 2020. *Python organization*. 7 January. Accessed March 2, 2020. https://docs.python.org/2/faq/general.html#what-is-python.

Shaik, Naushad. 2018. *5 reasons why learning Python is the best decision*. 21 September. Accessed March 2, 2020. https://medium.com/datadrivenin-vestor/5-reasons-why-i-learned-python-and-why-you-should-learn-it-as-well-917f781aea05.

Learning to Use Python: The Basic Aspects

Abstract Python is not a complex language, but it is imperative to understand how to program different commands based on the necessities of the user. This chapter focuses on the understanding of numbers, the use of parenthesis, iterative programming, algebraic operations, managing data and loops. The chapter aims as a result to help the reader in understanding the different processes that Python has for managing data, understanding the importance of each process and how to choose which one will fit the best model.

Keywords Booleans · Lists · Algebraic operations · Loops

The present chapter is written as the basics for understanding Python for Finance. If you feel that you are acquainted with the use of python and the process by which it operates, then move to the next chapter, if not please elaborate each of the exercises in a Jupyter Notebook because learning by practicing is the easiest way for learning Python.

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_2

Understanding Numbers in Python

To use numbers in Python, the notation is simple, given that an equation can be computed as simple as follows:

- Entering a Number
  - In [1]: 200
  - Out [1]: 200

You can achieve this just by pressing enter once the number has been inserted into the line of command in the Jupyter Notebook.

- Addition in Python
  - In [1]: 200 + 200
  - Out [1]: 400
- Subtracting in Python
  - In [1]: 200 − 200
  - Out [1]: 0
- Multiplying in Python
  - In [1]: 200 * 200
  - Out [1]: 40000
- Dividing in Python
  - In [1]: 200/200
  - Out [1]: 1.0
- Square root in Python
  - In [1]: (−1) ** (0.5)
  - Out [1]: 6.123233995736766e−17+1j

It is important in this aspect to discuss the different numbers that Python supports:

- *Integers*: Known as *int* are whole numbers without decimal points such as the result of the 200 + 200 = 400. The integers can be long integers known as *long* and have unlimited precision and plain integers that are called *integers* and have at least a 32 bits precision.
- *Floating Real Values*: Known as *float* and it is written with a decimal point or a scientific notation. An example is the 200/200 = 1.0
- *Complex numbers*: Known as *complex* and is a combination of imaginary numbers and real numbers such as the example of the (−1)**(0.5) = 6.123233995736766e−17+1j (Python 2006)

The properties above can be applied to any number by writing the following code:

- In [1]: int(200/200)
- Out [1]: 1

- In [1]: float(200−200)
- Out [1]: 0.0

- In [1]: complex(200+200)
- Out [1]: 400+0j

This is extremely useful when there is a necessity of changing the numbers into a format that is more useful when the necessity arrives.

As seen in the example before, the way to create a square root or to a power is through the double asterisk (**) just changing the number after the double asterisk into a decimal or an integer. Asterisk is useful also for packing arguments into a specific function or to use in positional arguments using keys (Hunner 2018).

- To a power of three
  - In [1]: (2)**(3)
  - Out [1]: 8

- The square root
  - In [1]: (4)**(0.5)
  - Out [1]: 8

Understanding Numbers in Python

During the present chapter, the *bitcoin-usd* data will be used as an example. To add this information, follow the process:

```
pip install ffn

import ffn

bitcoin = ffn.get('BTC-USD:Close', start=2021-01-01', end='2021-01-01')
bitcoin.tail()
```

btcusdclose	
Date	
2021-01-01	29374.152344
2021-01-02	32127.267578

To use numbers in Python, the notation is simple, given that an equation can be computed as simple as follows:

- Entering a Number
  - In [1]: 200
  - Out [1]: 200

You can achieve this just by pressing enter once the number has been inserted into the line of command in the Jupyter Notebook.

- Addition in Python
  - In [1]: 200+200
  - Out [1]: 400
- Subtracting in Python
  - In [1]: 200 − 200
  - Out [1]: 0
- Multiplying in Python
  - In [1]: 200 * 200
  - Out [1]: 40000
- Dividing in Python
  - In [1]: 200/200
  - Out [1]: 1.0
- Square root in Python
  - In [1]: (−1) ** (0.5)
  - Out [1]: 6.123233995736766e−17+1j

It is important in this aspect to discuss the different numbers that Python supports:

- Integers: Known as *int* are whole numbers without decimal points such as the result of the 200 + 200 = 400. The integers can be long integers known as *long* and have unlimited precision and plain integers that are called *integers* and have at least a 32 bits precision.

- Floating Real Values: Known as *float* and it is written with a decimal point or a scientific notation. An example is the 200/200 = 1.0
- Complex numbers: Known as *complex* and is a combination of imaginary numbers and real numbers such as the example of the (−1)**(0.5) = 6.123233995736766e−17+1j (Python 2006)

The properties above can be applied to any number by writing the following code:

- In [1]: int(200/200)
- Out [1]: 1

- In [1]: float(200 − 200)
- Out [1]: 0.0

- In [1]: complex(200 + 200)
- Out [1]: 400+0j

In the case of *bitcoin-usd* the numbers are as follow:

btcusdclose

Date	
2021-01-01	29374.152344
2021-01-02	32127.267578

Given that the numbers have decimal points, the type of number is a *float*. When analyzing using the command *type* in bitcoin the result is as follows:

```
type(bitcoin)


pandas.core.frame.DataFrame
```

This is extremely useful when there is a necessity of changing the numbers into a format that is more useful when the necessity arrives.

As seen in the example before, the way to create a square root or to a power is through the double asterisk (**) just changing the number after the double asterisk into a decimal or an integer. Asterisk is useful also for packing arguments into a specific function or to use in positional arguments using keys (Hunner 2018).

- To a power of three
  - In [1]: (2)**(3)
  - Out [1]: 8

- The square root
  - In [1]: (4)**(0.5)
  - Out [1]: 8

The following can be done with the variable *bitcoin*. Given that there are two values in the DataFrame (which will be explained in future chapters), the following process will choose the latest value:

```
bitcoin[1:]
```

	btcusdclose
Date	
2021-01-02	32127.267578

If the price of the bitcoin in USD is needed to the power of three the process is as follows:

```
bitcoin[:1]**3
```

	btcusdclose
Date	
2021-01-02	2.534522e+13

And for the square root:

```
bitcoin[:1]**0.5
```

	btcusdclose
Date	
2021-01-02	171.388892

Using Data Structures in Python

When working with financial data, it is extremely important to know how to adequate the data based on the process that it will be used. The purpose of this section is to help understand the most important process that can be done with data.

What Is a List?

Lists are one of the most important features in Python because it helps in managing a data frame. When using lists there are certain questions that are important to answer one by one and it is what will be done in each chapter.

A list equivalent is an array, they are both objects which contain information. There is a difference when comparing a list or an array to a string, because a string cannot be altered, and it is useful when the information in the array will be used to guarantee that the data will be same during the duration of the program. It is not that useful in finance, since the data that is aimed for has to change continuously but it is important to understand the properties that each has.

How to Create a List?

To create a list there are two useful items, (1) the brackets ([ ]) and (2) the comma (,). Both of them are needed when creating a list first because the brackets will allow to define the space of the array, meaning that it will contain the elements. An example below:

- In [1]: my_list = [1,2,3]
- Out [1]: [1,2,3]

There is no limit when adding elements to a list. There is also no restriction when adding to the array a *float*, a *complex* or an *integer* (Python 2020). An example below:

- In [1]: my_list = [1,2,3,0.5,400+0j]
- Out [1]: [1,2,3,0.5,400+0j]

When adding a letter or a word, one has to add the quotes for adding them to a list. An example below:

- In [1]: my_list = [1,2,3,0.5,400+0j, "apple", "book", "b"]
- Out [1]: [1,2,3,0.5,400+0j, "apple", "book", "b"]

For the following example, two of the most important index will be used with the following code:

```
spy = ffn.get('^GSPC:Close', start='2021-01-01', end='2021-02-01')


dow = ffn.get('^DJI:Close', start='2021-01-01', end='2021-02-01')


list = [spy[18:],dow[18:]]
```

The problem with this process is that given that the information comes from a DataFrame, the information is not presented properly given the structure of the information.

```
gspcclose
 Date
 2021-01-29  3741.26001,                djiclose
 Date
 2021-01-29  30170.699219]!
```

Measuring a List

To know how many elements my list has, the function to be used is len(). Len() will allow us to understand the elements inside the list in order to create calculations. An example as follows:

- In [1]: my_list = [1,2,3,0.5,400+0j, "apple", "b"]
- Out [1]: [1,2,3,0.5,400+0j, "apple", "b"]

- In [1]: len(my_list)
- Out [1]: 7

Since the list has been modified to add different elements, the total of elements is seven (7). For example, in my_list=[10, 10.25, "string"] there are three elements, if len(my_list) is used the result will be three (3) because of each element.

In the example using the Dow Jones Industrial Average and Standard and Poor 500, the same can be applied.

```
len(list)
```

```
2
```

The same can be used in a DataFrame, for example in the variable *bitcoin* the process can be followed:

```
len(bitcoin)
```

```
2
```

Given that the variable *bitcoin* only had two prices, the length is two (2).

Indexing and Cutting a List

Indexing is very useful for knowing where the data is located and doing processes with particular data. In finance, this is a very interesting function because returns can be calculated on a specific date or a simple moving average based on a specific number. An example as follows:

- In [1]: my_list[1]
- Out [1]: 2

One of the most important aspects when using list is that lists in Python, as many other computation languages, starts by counting from zero (0). This is why when indexing the element in position one [1] the result

is the number two, because it is in position one ([1,2,3,0.5,400+0j, "apple", "b"]).That is one of the most important aspects of using the function len() because without knowing how many elements are in a list, it is difficult to understand which position belongs to each element.

In a list there can also be negative indexing, which means that a negative number can be used for recalling an element in a list. An example below:

- In [1]: my_list[−1]
- Out [1]: 'b'

If the process is done backward by using a negative number, then the result is the letter 'b' because as it is the last element [1,2,3,0.5,400+0j, "apple", 'b']).Indexing can also be used to recall from a specific number onward by using the colon (:). An example below:

- In [1]: my_list[1:]
- Out [1]: [2,3,0.5,400+0j, "apple", "b"]).

Before, the example was demonstrated with the Dow Jones and the S&P 500 by cutting the list as follows to get the last value:

```
list = [spy[18:],dow[18:]]
```

The example below takes all the elements except the element in the first position, in the position of zero (0). It can also be used for indexing between different values. An example below:

- In [1]: my_list[3:5]
- Out [1]: [0.5,400+0j]

In this case it includes position three (0.5) and the position four (400+j) but it excludes the word "apple" because it is not included.

In the Dow Jones and S&P 500 example, the process can be elaborated in the same way as before with the problem of the arrangement:

```
list = [spy[17:19],dow[17:18]]


[              gspcclose
 Date
 2021-01-28  3787.379883
 2021-01-29  3741.260010,                djiclose
 Date
 2021-01-28  30603.359375]
```

If the fifth element wants to be included then the most useful process is to add the sixth position as follows:

- In [1]: my_list[3:6]
- Out [1]: [0.5,400+0j, 'apple']

List can also be used to recall up to a certain element. In this case, the process is similar to the one above, but the only element used will be after the colon. An example as follows:

- In [1]: my_list[:5]
- Out [1]: [1,2,3,0.5,400+0j]

As before, with Dow Jones and S&P 500 example, the process can be elaborated in the same way:

```
list = [spy[:5],dow[:5]]


      gspcclose
 Date
 2021-01-04  3700.649902
 2021-01-05  3726.860107
 2021-01-06  3748.139893
 2021-01-07  3803.790039
 2021-01-08  3824.679932,                djiclose
 Date
 2021-01-04  30223.890625
 2021-01-05  30391.599609
 2021-01-06  30829.400391
 2021-01-07  31041.130859
 2021-01-08  31097.970703]
```

Both processes done before can also be done by using negatives, an example is as follows:

- In [1]: my_list[:−2]
- Out [1]: [1,2,3,0.5,400+0j]

Or:

- In [1]: my_list[−2:]
- Out [1]: ["apple", "b"]

A list can also be added with other list, this is an important feature when trying to unite two arrays. An example as follows:

- my_list = [1,2,3,0.5,400+0j, "apple", "b"]
- my_list_2 =["oranges",5,0,5,300+1]

- In [1]: my_list + my_list_2
- Out [1]: [1,2,3,0.5,400+0j, "apple", "b","oranges",5,0,5,300+1]

With just an addition sign (+) two lists were concatenated. This is useful when created having two sets of data that will be used conjointly.

This can also be used for adding elements:

- In [1]: my_new_list = my_list + my_list_2 + ['Python']
- Out [1]: [1, 2, 3, 0.5, (400+0j), 'apple', 'b', 'oranges', 5, 0.5, (300+1j), 'Python']

A list can also be duplicated by multiplying it by two:

- In [1]: my_list*2
- Out [1]: [1, 2, 3, 0.5, (400+0j), 'apple', 'b', 1, 2, 3, 0.5, (400+0j), 'apple', 'b']

In the list created that involved the indexes, the process can be done as follow for multiplication, with the problem that it even duplicates the titles in a DataFrame:

```
list * 2
```

```
     gspcclose
 Date
 2021-01-29  3741.26001,                djiclose
 Date
 2021-01-29  30170.699219,              gspcclose
 Date
 2021-01-29  3741.26001,                djiclose
 Date
 2021-01-29  30170.699219]
```

If the process is done in the DataFrame by addition, the following answer will appear as a type error.

```
list + 2
```

```
---------------------------------------------------------
-
TypeError                                 Traceback (most recent call
last)
<ipython-input-76-c14dd7274c88> in <module>()
----> 1 list + 2

TypeError: can only concatenate list (not "int") to list
```

The reason for this type of error is that, as explained before a not integer given that the information is required from an API, in this case *Yahoo Finance*. It could be suggested that for conversion one should use the *int* command, but the error is as follow:

```
TypeError                                 Traceback (most recent call
last)
<ipython-input-77-66c0276b75cb> in <module>()
----> 1 int(list)

TypeError: int() argument must be a string, a bytes-like object or a
number, not 'list'
```

Appending Lists

Another method of adding elements to a list is by the *append( )* that is useful for adding data. Although it can be done by using the addition sign (+), the *append( )* method is used in data structures and it could behoove the user when using databases.

- In [1]: my_list.append("Python")
- Out [1]: [1, 2, 3, 0.5, (400+0j), 'apple', 'b', 'Python']

Apart from a word, another list of items can be appended into the original list. An example below:

- In [1]: my_list.append(my_list_2)
- Out [1]: [1, 2, 3, 0.5, (400+0j), 'apple', 'b', 'Python', ['oranges', 5,0.5, (300+1j)], ['oranges', 5, 0.5, (300+1j)]]

This is useful when creating a new list with data from other sets. In the example, as follows.

Arranging Lists

A list can be arranged to suit the needs of the user. This is extremely useful when the list has to be arranged from first to last or it has to be sorted by a certain requirement. An example as follows with the list my_list=[1,2,3]

- In [1]: my_list.reverse()
- Out [1]: [3, 2, 1]

Another procedure for arranging list is sorted, which arranges the data from first to last. This is helpful when the data has to be arranged, for example, from the smallest return to the highest return. An example as follows with the list my_list=[4,1,4,3,2,10,20,23]

- In [1]: my_list.sort()
- Out [1]: [1, 2, 3, 4, 4, 10, 20, 23]

An important aspect of this is that the data, once it is sorted this is a permanent aspect of the list. Therefore, if the data that is being used

belongs to the original, the best advice is to create a new list with the original data that can be modified.

From List to Matrices

To *nest* a list is to arrange data into a matrix. This is an easy procedure because of the capacity that Python has for modifying data. An example below for the understanding of the use of nesting with the following lists:

a_1=[1,2,3]
a_2=[4,5,6]
a_3=[7,8,9]

- In [1]: matrix=[a_1, a_2, a_3]
- Out [1]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

There is certain procedure that can be executed when using matrix that are similar to a list. An example below:

- In [1]: matrix=[0]
- Out [1]: [[1, 2, 3]

In the example above the data that is being retrieved is the first list. If the first element of the list will be retrieved the process is as follows:

- In [1]: matrix=[0][0]
- Out [1]: 1

The first bracket is for retrieving the list and the second bracket is for the first element in the list. This is based on the fact that in Python the position zero (0) is the first position.

From List to Dictionaries

A dictionary is an element for handling data that is different to strings and list. For a dictionary a different programming element is used: the curly brackets {}. Dictionaries are very flexible, and they are useful when handling data because it gives a sense of order that is helpful when the data is from a big set.

A dictionary is divided into two elements: (1) the keys and (2) the values. The keys are useful because the key will determine the value that is being requested. The key is separated from the value by a colon (:) to indicate that the first element is the key and the second element is the value. An example as follows:

```
my_dictionary = {'key1': 'value1', 'key2': 'value2', 'key3':'value3'}
```

- In [1]: my_dictionary['key1']
- Out [1]: 'value1'

A dictionary can handle different types of data such a integers, floats, complex or long. This is one of the most flexible characteristics that a dictionary has:

```
my_dictionary_2 = {'a': 23.50, 'b': 27982, 'c':'hola'}
```

- In [1]: my_dictionary_2 ['a']
- Out [1]: 23.5

Another important aspect of dictionaries is that it can perform arithmetic operations with the data contained in the keys.

- In [1]: my_dictionary_2 ['a'] − 23
- Out [1]: 0.5

- In [1]: my_dictionary_2 ['a']*10
- Out [1]: 235

- In [1]: my_dictionary_2 ['a']+23.50
- Out [1]: 47.0

- In [1]: my_dictionary_2 ['a']/2
- Out [1]: 11.75

This is an important feature because it does not alter the data. Each calculation is created apart from the original data, creating results without any alteration.

Modifying a Dictionary

There will be moments when it will be necessary to alter the information inside the dictionaries. It is a really simple process.

- In [1]: my_dictionary_2 ['a'] = my_dictionary_2 ['a'] + 100
- Out [1]: {'a': 123.5, 'b': 27,982, 'c': 'hola'}

In, the example above 100 was added to the first key and modified the value for a result of 123.5 This procedure can be done with addition, multiplication, division, square root or any other arithmetic approach. An example of this is as follows:

```
d={}
d['price']= '$25000'
d['car']= 'Peugeot'
```

- In [1]: d
- Out [1]: {'car': 'Peugeot', 'price': 25000}

If a discount of 10% should be applied to the price of the car the process would be as follows.

- In [1]: d['price']*(0.9)
- Out [1]: 22500.0

Other Interesting Functions of a Dictionary

As recalled before, the first element of the dictionary, the one that contains the data is retrieved by asking for the *key*. In this aspect, Python has the function key to know which are the keys that are included in the dictionary. An Example as follows:

```
countries={}
countries={'Guatemala': 'Guatemala', 'Costa Rica': 'San José',
'Nicaragua': 'Managua'}
```

- In [1]: countries.keys()
- Out [1]: dict_keys(['Guatemala', 'Costa Rica', 'Nicaragua'])

This allows the user to understand which keys have been chosen and how to retrieve the data in the list. For example, it can be the name of the stocks and the values that each stock holds. If the values of the dictionary are not known and want to be retrieved the process is as follows:

- In [1]: countries.keys()
- Out [1]: dict_values(['Guatemala', 'San José', 'Managua'])

The DataFrame

As seen before, the book is centered on creating DataFrames with data from the internet obtained directly from the webpage. For this, it is important to understand the use of a DataFrame and the possibilities that it has when compared to a list.

For the process of understanding a DataFrame, the *f.fn()* module will be used and the following stocks:

```
stocks = ffn.get('gme:Close, aapl:Close, msft:Close', start='2020-01-01',
end='2021-02-01')
stocks.head()
```

	gmeclose	aaplclose	msftclose
Date			
2020-01-02	6.31	75.087502	160.619995
2020-01-03	5.88	74.357498	158.619995
2020-01-06	5.85	74.949997	159.029999
2020-01-07	5.52	74.597504	157.580002
2020-01-08	5.72	75.797501	160.089996

To create a DataFrame a library named *pandas* is needed. In this case *pandas* stands for *panel data* which is basically using data in columns and rows in order to provide calculations. It also provides a better visualization when using data which makes it easier to edit.

Data_Frame =
pd.DataFrame({'col1':[1,2,3,4], 'col2':[200,300,200,300], 'col3':['abc', 'def', 'ghi', 'xyz']}).

- In [1]: Data_Frame
- Out [1]:

	col1	col2	col3
0	1	200	abc
1	2	300	def
2	3	200	ghi
3	4	300	xyz

A DataFrame has similar properties to a dictionary. For example, the columns can be recalled one by one depending on the information needed.

- In [1]: Data_Frame['col1']
- Out [1]:

```
0  1
1  2
2  3
3  4
Name: col1, dtype: int64
```

If there are values repeated such as in the second column, with the *unique* command the values can be removed.

- In [1]: Data_Frame['col2'].unique()
- Out [1]: array([200, 300])

In DataFrames, arithmetic functions can also be applied.

- In [1]: Data_Frame['col1'].sum()
- Out [1]: 10

- In [1]: Data_Frame['col1'].mean()
- Out [1]: 2.5

- In [1]: Data_Frame['col1'].multiply(5)
- Out [1]:

```
0   5
1  10
2  15
3  20
Name: col1, dtype: int64
```

The DataFrames can be altered to remove a column to create a new DataFrame with the *drop* command.

- In [1]: Data_Frame.drop('col1',axis=1)
- Out [1]:

	col2	col3
0	200	abc
1	300	def
2	200	ghi
3	300	xyz

By choosing any of the column the specific column is removed in the DataFrame but the change is not permanent which behooves the user when trying to undo the process of calculation. DataFrames can also be sorted just as the dictionaries were sorted:

- In [1]: Data_Frame.sort_values('col2')
- Out [1]:

col1	col2	col3	
0	1	200	abc
2	3	200	ghi
1	2	300	def
3	4	300	xyz

DataFrames can also be used with Booleans. This is one of the most interesting features. An example as follows:

- In [1]: Data_Frame >200
- Out [1]:

	col1	col2	col3
0	False	False	True
1	False	True	True
2	False	False	True
3	False	True	True

	col1	col2	col3
0	1	200	abc
1	2	300	def
2	3	200	ghi
3	4	300	xyz

As a Boolean, the command to search those values that are bigger than 200. As an interesting aspect, when the comparison is based on letters, the Boolean response is for it to be true, but this is based on comparing two elements that are different.

When using the DataFrame with the *Yahoo Finance* API, there are important differences that are useful when working with data. For now, the following exercise is the application of what was demonstrated before:

– Choosing a column

```
stocks['gmeclose']
```

```
Date
2020-01-02      6.310000
2020-01-03      5.880000
2020-01-06      5.850000
2020-01-07      5.520000
2020-01-08      5.720000
                  ...
2021-01-25     76.790001
2021-01-26    147.979996
2021-01-27    347.510010
2021-01-28    193.600006
2021-01-29    316.644714
```

– Choosing unique values

```
stocks['aaplclose'].unique()
```

```
array([ 75.08750153,  74.35749817,  74.94999695,  74.59750366,
        75.79750061,  77.40750122,  77.58249664,  79.23999786,
        78.16999817,  77.83499908,  78.80999756,  79.68250275,
        79.14250183,  79.42500305,  79.80750275,  79.57749939,
        77.23750305,  79.42250061,  81.08499908,  80.96749878,
        77.37750244,  77.16500092,  79.71250153,  80.36250305,
        81.30249786,  80.00749969,  80.38749695,  79.90249634,
        81.80000305,  81.21749878,  81.23750305,  79.75      ,
        80.90499878,  80.07499695,  78.26249695,  74.54499817,…)]
```

– Using *sum* with a column

```
stocks['msftclose'].sum()
```

```
53055.86494445801
```

– Using *mean* with a column

```
stocks['msftclose'].mean()
```

```
195.05832700168386
```

– Using *multiply* with a column

```
stocks['msftclose'].multiply(5)
```

```
Date
2020-01-02     803.099976
2020-01-03     793.099976
2020-01-06     795.149994
2020-01-07     787.900009
2020-01-08     800.449982
                  ...
2021-01-25    1147.649994
2021-01-26    1161.650009
2021-01-27    1164.499969
2021-01-28    1194.649963
2021-01-29    1173.424988
Name: msftclose, Length: 272, dtype: float64
```

– Using *drop* to remove a column

```
stocks.drop('msftclose', axis=1)
```

	gmeclose	aaplclose
Date		
2020-01-02	6.310000	75.087502
2020-01-03	5.880000	74.357498
2020-01-06	5.850000	74.949997
2020-01-07	5.520000	74.597504
2020-01-08	5.720000	75.797501
...	...	...
2021-01-25	76.790001	142.919998
2021-01-26	147.979996	143.160004
2021–01-27	347.510010	142.059998
2021-01-28	193.600006	137.089996
2021-01-29	316.644714	133.184998

272 rows × 2 columns

– Sort values

```
stocks.sort_values('gmeclose')
```

	gmeclose	aaplclose	
Date			
2020-04-03	2.800000	60.352501	153.830002
2020-04-02	2.850000	61.232498	155.259995
2020-04-06	3.090000	65.617500	165.270004
2020-04-01	3.250000	60.227501	152.110001
2020-04-07	3.270000	64.857498	163.490005
...	...	...	...
2021-01-25	76.790001	142.919998	229.529999
2021-01-26	147.979996	143.160004	232.330002
2021-01-28	193.600006	137.089996	238.929993
2021-01-29	316.644714	133.184998	234.684998
2021-01-27	347.510010	142.059998	232.899994

272 rows × 3 columns

– Using a Boolean

```
stocks > 140
```

Date	gmeclose	aaplclose	msftclose
2020-01-02	False	False	True
2020-01-03	False	False	True
2020-01-06	False	False	True
2020-01-07	False	False	True
2020-01-08	False	False	True
...	...	...	...
2021-01-25	False	True	True
2021-01-26	True	True	True
2021-01-27	True	True	True
2021-01-28	True	False	True
2021-01-29	True	False	True

272 rows × 3 columns

Boolean, Loops and Other Features

Booleans are operators that allow us to know if the element in a function is TRUE or FALSE. Similar to the example above, the process allowed us to know which values are bigger than 200. Booleans are extremely important when analyzing data because they allow for decision-making in finance.

- In [1]: 1 > 2
- Out [1]: False

- In [1]: 1 < 2
- Out [1]: True

This is one of the most important aspects when comparing data. For example, if there is a need to know if the Value at Risk (VaR) will be higher than a specific percent or number, the Boolean could be useful to understand the process.

To understand how Booleans work, it is important to specify the different cases in which a Boolean creates a response that can be useful for the user (Table 1).

Booleans can also be used as a chain of comparison. When there are different elements included, the Booleans can specify if the sequence is correct. An example below:

Table 1 Understanding operators

Operator	*Description*	*Example*
==	If the values of the operators are equal the result will be TRUE, if not the result will be false	2==1 (FALSE) 1==1 (TRUE)
!=	This operator is the contrary of the (==) sign because it means not equal. When using the != the order of the exclamation mark (!) and the equal sign (=) has to be as shown in the example There is another way of computing the same operator with the same results, but it is less useful. The use is <>	2 != 1 (TRUE) 1 != 1 (FALSE)
>	This operator specifies that the argument on the left is bigger than the argument on the right	2 > 1 (TRUE) 1 > 2 (FALSE)
<	This operator specifies that the argument on the left is smaller than the argument on the right	2 < 1 (FALSE) 1 < 2 (TRUE)
<= o >=	This operator is known as equal or greater than, or equal or lesser than. This will include the argument that is equal, being the main difference between the greater than and lesser than operators	3 >= 3 (TRUE) 3 >= 4 (FALSE) 4 >= 3 (TRUE)

Source Elaborated by the author

- In [1]: 1 < 2 < 3
- Out [1]: True

Or:

- In [1]: 1 > 10.4 < 3
- Out [1]: False

For Booleans the use of the command (and) or (or) is extremely useful when comparing equations. This allows the user to understand if various statement is True or False. An example as follows:

- In [1]: 3 > 2 and 3 < 4
- Out [1]: True

Or:

- In [1]: 3 != 2 and 3 == 2
- Out [1]: False

When using (or), these questions if one statement is correct based on the other statement. It will tell us if one of the statements is True regarding if the other statement is False.

- In [1]: 3 < 2 or 1 == 1
- Out [1]: True

Another example:

- In [1]: 3 < 2 or 1 != 1
- Out [1]: False

If, Else and Elif in Python

In Python there are three important statements, (1) if, (2) else and (3) elif. Each of them has a proper way to be used and in which situation it should be used.

If:
It is used when the user wants there to be an alternative to what is presented. This is, when the user chooses an action if a case happens.

The Syntax for writing an *if* statement is as follows:

```
if example:
performs action1
```

Example:

- In [1]: if True:
  print ('It is correct')
- Out [1]: It is correct

Else:

The *else* statement is useful when combined with the *if* statements. The reason for this is because when we use the *if* function, we are asking that if something happens, the program should demonstrate a certain result. With *else*, we complement the *if* function and tell the function to who what happens if what is commanded is not fulfilled. In this way we can have the two parts of the equation, what happens and what does not happen.

Example:

- In [1]: a = False

```
if a:
    print ('a is correct')
else:
    print ('anything but correct')
```

- Out [1]: anything but correct

In this example, what can be observed is that Python understands the True and the False commands implicitly. These commands that were previously identified as Boolean, allow the user to establish a relationship. In the previous case, the variable *a* was part of the function and the value False placed a conditional giving as a result that if *a* is True for it to print *a is correct* and the occasion that the result is otherwise the program will print *a is anything but correct*. For the value that is True, the following code is as follows:

- In [1]: a = True

```
if a:
    print ('a is correct')
else:
    print ('anything but correct')
```

- Out [1]: a is correct

Elif:

The *elif* statement is used when there are different arguments. In the case of an *elif* statement what the program wants to validate is if the variables are True or False. An example of an *elif* statement is as follows:

```
if expression:
statement(a)
elif expression(2):
statement (a)
elif expression3:
statement (a)
elif expression4:
statement(a)
```

The following example takes into account the theory of the Value at Risk (VaR). The purpose of the elif statement is to acknowledge if the fall of the VaR is one or the other. In this case, the values proposed are in million. To know more about the VaR, please refer to the chapter titled Value at Risk (VaR).

```
In [1]:
var = 100
if var == 200:
        print ("1 – Value is True")
        print (var)
elif var == 150:
        print ("2 – Value is True")
        print (var)
elif var == 100:
        print ("3 – Value is True")
        print (var)
else:
        print ("4 – Value is False")
        print (var)

Out [1]: 3 – Value is True
   100
```

Loops

There are two types of loops that are extremely useful in Python. The first is a *for loop* and the second one is a *while loop*. Each of the loops has its own structure and should be used base on the necessity of the program.

For Loop

Loops are iterations that allow the user to elaborate sequences and use them once and again. For creating a *for loop* the process although complicated is centered on the purpose of the iteration.

Example:

```
m = [1,2,3,4,5,6]
```

```
In [1]:
for x in m:
    print (x)

Out [1]:
1
2
3
4
5
6
```

The iteration of the *for* loop allows us in a simple way to separate each of the elements included in the list. If a *for loop* was not used the process would have been extremely complicated and also time consuming. An example of the process without the *for* loop would have been as follows:

```
print (m[0])
print (m[1])
print (m[2])
print (m[3])
print (m[4])
print (m[5])
```

By using the iteration, it is not only simple but also easy to do a process with less time. This will be useful when elaborating Monte Carlo simulations or for applying the Sharpe Ratio into a Markowitz model.

Python can also engage in an iteration or a loop without having to define the element. For example, in the *for loop* the *x* was used and it did not contain an element, this is basic because when the loop ends its iteration this is the out result. If in the loop the *x* is changed into a word such as example, the result is the same.

```
In [1]:
for example in m:
    print (example)
```

```
Out [1]:
1
2
3
4
5
6
```

Another interesting thing is that when a string is used with a combination based on a variable, the result is the replication of the string. This is useful for understanding the process of a *for loop*.

```
m=[1,2,1,2,1]
```

```
In [1]:
for x in m:
    print ('I am a result')
```

```
Out[1]:
I am a result
I am a result
I am a result
I am a result
I am a result
```

In the example above it is important to understand that it is base on the elements inside the variable *m* that amount to five (5) and not on the numbers that are in the variable [1,2,1,2,1].

Now an important element is the modulo. The modulo allows us to obtain the waste and is identified by the percent sign (%). We can use this in the following way.

If the user divides 12 into 5, the result will be that 5 fits twice inside 12 but there is a remainder of 2. In Python the solution can be found in the following way:

```
In [1]: 12%5
Out[1]: 2
```

One of the usual exercises that are practiced when using the *modulo* function is the identification of even numbers. Knowing that the

function *modulo* (%) gives us the residue, what we must do to find even numbers is to divide within 2 and the result to be zero (0). An example as follows:

```
m=[1,2,3,4,5,6]

In [1]:
for number in m:
        if number%2 == 0:
                print (number)

Out[1]:
                2
                4
                6
```

If the example is with odd numbers, the process would only have to change the part that indicates if number%2 == 0: and change it by if number%2 == 1:

To the iteration *else* can be added to operation with a string as a response. An example is the following:

```
In [1]:

for number in m:
if number%2 == 0:
print ("the number es even")
else:
print ("the number is uneven")

Out[1]:

the number es even
the number es even
the number es even
the number is uneven
```

In the same way the arguments can be changed to obtain a specific string or another element. An example as follows:

```
In [1]:
for number in m:
        if number%2 == 1:
                print ("The number is even")
else:
        print (number)
```

Out[1]:

```
1
The number is even
3
The number is even
5
The number is even
```

When a loop is written it is important to remember that what is inside the loop is still being iterated over and over again. Therefore, print is usually left out of the equation. An example is the following:

```
m = [1,2,3,4,5,7,8,9,10]

add = 0
```

In [1]:

```
for number in m:
        add = add + number
print (add)
```

Out[1]: 49

What the previous interaction does is add the elements of the list *m* one after the other to obtain 49. In this case, the *for loop* is separated from the print and the determination of the variable because the iteration must be individual, but it will take the other elements.

For loops can also be used for strings with the same modality that are used for elements.

In [1]:

```
a= "Letter"
for letter in a:
 print (letter)
```

Out[1]:

```
L
e
t
t
e
r
```

When the process is done in a dictionary there are other elements to consider. It is important to remember that when making a dictionary the keys are defined for each of the elements in the list. Therefore, the definition must be specific for the loops.

In [1]:

```
dictionary = {'k1':1, 'k2':2, 'k3':3}

for element in dictionary:
    print (element)
```

Out[1]:

```
k1
k2
k3
```

To use this in Python 3 the items () command allows us to perform the function. An example as follows:

In [1]:

```
for k,v in dictionary.items():
        print (k)
        print (v)
```

Out[1]:

k1
1
k2
2
k3
3

When creating a portfolio, the method of a *for loop* will be used to group different tickers. The process is as follows:

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)

tickers = ['NFLX', 'DIS', 'TSLA', 'AMZN']
stocks = pd.DataFrame ()
for x in tickers:
    stocks[x] = web.DataReader(x, 'yahoo', start, end)['Close']
```

While Loop

The *while loop* will give a result until there is a true value (True). That is to say that it will be repeated again and again until obtaining the said result. This can be quite useful to determine a value or a string.

In [1]:

```
a=0

while a<10:
print ("a is", a)
a+=1

else:
print ("end")
```

Out[1]:

```
a is 0
a is 1
a is 2
a is 3
a is 4
a is 5
a is 6
a is 7
a is 8
a is 9
end
```

In this case what is being checked is that the numbers less than 10 will be added to one (1) until it is less than 10. When it reaches the tenth (10) value, it will print a phrase that says "end". In this case we can verify the existence of the elements less than 10 other than being able to add a data.

To work with while loops it is important to know 3 commands:

1. break,
2. continue and
3. pass

Break allows the user to close the loop that is running. Continue allows the user to do one more iteration and pass allows us to leave the loop without effect. While loops can be longer, including more functions to precisely define what we want.

In [1]:

```
a=0

while a<5:
print ("a is", a)
print ("a is less than 5")
a+=1
```

```
if a==4:
print ("a is equal to 4")
else:
print ("Continue")
continue
```

Out[1]:

```
a is 0
a is less than 5
We continue
a is 1
a is less than 5
We continue
a is 2
a is less than 5
We continue
a is 3
a is less than 5
a is equal to 4
a is 4
a is less than 5
continue
```

What was done in this code is the following.

1. The variable "a" was created and a value of zero (0) was placed.
2. A while loop was created in which when "*a*" was less than (<) 5 then it would print "*a is*" and that could be the value of "*a*". Then it would print "a is less than 5".
3. An *if conditional* was placed to say that if the "*a*" was absolutely equal (==) to four then it would print "a is equal to 4." For all the values that are not absolutely equal to four, the string "follow" would be printed.
4. The "continue" command is placed to give it iteration, that is, continue replicating.

When we use the break function, we put an end to it. That is why it is important to use it correctly because it will interrupt the loop.

In [1]:

```
a=0

while a<5:
print ("a is", a)
print ("a is less than 5")
a+=1

if a==2:
print ("a is equal to 2")
break
else:
print ("We continue")
continue
```

Out[1]:

```
a is 0
a is less than 5
We continue
a is 1
a is less than 5
a is equal to 2
```

What was indicated in the code is that it stops once it reaches the value that is absolutely equal to 2. Therefore, when the number 2 is presented, the last line reads "a is equal to 2" and the code does not continue. This despite having an "else" that should continue the code.

Finally, if one writes *pass* result would be the same result as in the loop without break because it is an argument so that nothing happens.

List Comprehension

The comprehensions of lists allow us to easily and clearly make a list with a different notation and that this already includes a loop, specifically a for loop. It is easy to understand as follows:

In [1]:

```
m=[]
```

```
for letter in "word":
   m.append(letter).
print (m)
```

Out[1]:

['w', 'o', 'r', 'd'].

In this case it could have been done with a list with the variable "m" through indexing "word" and that each of the letters was an element. For example, if one uses m [0] it will give the letter "w" or if we use m [3] it would give us the letter "d".

The above can be developed in a simple manner as follows:

```
n=[word for word in "word"]

print (n)
```

Result

['w', 'o', 'r', 'd']

The lists can also be used for mathematical developments. An example is that one would like to obtain the square of a list between 0 and 11. For that, it would be tedious to have to go element by element, developing it. In this case the simplest form is the following:

In [1]:

```
list = [x**2 for x in range (0,11)].
print (list).
```

Out[1]:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

The above can be decompressed as follows. First the variable "list" was created and assigned to multiply the data within the range of 0 to 11 high (**) squared. In this sense we obtain that 3 by 3 is equal to 9 or

that 10 by 10 is equal to 100. It is to remember that it reaches number 10 because the ranges do not include the last data.

Previously, it had been analyzed to obtain even results of an equation. With the lists can be done in a simple way as follows:

In [1]:

```
list2 = [number for number in range (20) if number%2 == 0]
print (list2)
```

Out[1]:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The above is simple because it creates a loop using the function modulo (%) that indicates the values that have zero residue when being divided by 2 then they are printed. Since the pairs are the only ones that in a division of 2 can have zero residue, that is why it is easy to achieve the above.

References

Hunner, Trey. 2018. *Trey Hunner*. 11 October. Accessed March 23, 2020. https://treyhunner.com/2018/10/asterisks-in-python-what-they-are-and-how-to-use-them/.

Python. 2006. *2.3.4 Numeric types -- int, float, long, complex*. 18 October. Accessed March 23, 2020. https://docs.python.org/2.4/lib/typesnumeric.html.

Python. 2020. *Data structures*. 23 March. Accessed March 23, 2020. https://docs.python.org/3/tutorial/datastructures.html.

Using FRED® API for Economic Indicators and Data (Example)

Abstract There are different application programming interfaces (API) for accessing data directly from the web and one of the most important for economic and financial analysis is the Federal Reserve of Saint Lois (FRED).

Keywords Data · Inflation · Growth · Deflator

For the application to indicators, first, the API (application programming interface) of the Federal Reserve of Saint Lois (FRED) will be used to retrieve information The FRED is an important database for macroeconomic concepts that are necessary for understanding the market. The example will be developed in *Google Colaboratory* to exemplify how to create a notebook that can be shared.

Installing the FRED® API

The FRED® API tool is easy to use and it leads to a faster analysis of the macroeconomic situation.

For installing the API, one should visit the webpage https://fred.stlouisfed.org/docs/api/fred/ or google search the term FRED® API to obtain the result. For the use of the FRED API, it is important to ask for a key, the key is useful to the FRED because it gives representation

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_3

to the person using the API. To obtain the key one should visit the API Keys page at https://research.stlouisfed.org/docs/api/api_key.html. The key can be requested by creating a user and filling out the information. Once the key is received (it is usually a quick process), then it can be used for analyzing the data.

Using the FRED® API to Retrieve Data

The first process for using the FRED ® API is to install it in the environment that will be used. In this example, the Google Collaboratory will be used given that it is a platform that can be used in similarity with the use of Anaconda. For installing the FRED® API the process is as follows:

First Step

```
pip install fredapi
```

The first process is using pip install[1] which allows to retrieve the information of the FRED. In the present example by using Google Collaboratory the process is extremely simple.

Second Step

```
from fredapi import Fred
fred = Fred(api_key='XXXXXXXX')
```

In this step, once the fredapi is installed then the Fred data library can be retrieved. For this it is necessary to have the API_key which will allow the retrieval of the information.

Third Step

To retrieve information from the FRED one should know how to use the FRED website and how to retrieve the information. For the following exercise will use the information from the FRED website which is as follows: https://research.stlouisfed.org/.

[1] For more information concerning pip install please visit: https://pip.pypa.io/en/stable/reference/pip_install/.

The search bar is useful for searching the information which is needed. In this example the term *economic growth* will be used, once entered the option of *search only FRED economic data* will be chosen. This filters the result in the data that can be used as an API.

The result page based on the search will return different economic indicators which can be filtered based on the concepts such as:

- Indexes
- Prices
- Price Index
- Consumer Price Index
- Employment

The information can also be filtered by geography types, geographies, frequencies, sources, releases and seasonal adjustments. For this example, the refine of the search won´t be used and the first result on the search will be the example.

When selected the result *Consumer Price Index: Total All Items for the United States* the data is illustrated by the FRED in a line graph. Next to the name of the *Consumer Price Index: Total All Items for the United States* there is a code in parenthesis `CPALTT01USM657N`. This code is useful for retrieving the information.

It is important to visit the indicator on the webpage to understand the frequency (in this case it is monthly), the last observation and last update as well as the units.

In the Google Collaboratory or Anaconda, with the code, it is possible to retrieve the information. To retrieve the data the process is as follows:

```
consumer_price_index = fred.get_series('CPALTT01USM657N')
```

The variable *consumer_price_index* has been created and now can be used for descriptive analysis or to develop an econometric model. The command tail() can be used to analyze the latest information as follows:

```
consumer_price_index.tail()
```

```
2020-05-01    0.001950
2020-06-01    0.547205
2020-07-01    0.505824
2020-08-01    0.315321
2020-09-01    0.139275
dtype: float64
```

There is a second method which is easier and works better with the methods that are going to be seen in this book.

The first step is to install de FRED API:

```
pip installfredapi
```

The command *pip*[2] is the program that installs the packages in Python. Once the *fredapi* is installed the following process is to install *pandas_datareader*. With *pandas_datareader* it is possible to access different information such as FRED, World Bank, OECD and NASDAQ. For more information concerning the different sources and how to use them please visit: https://pandas-datareader.readthedocs.io/en/latest/remote_data.html.

```
import pandas_datareader as pdr
```

After *pandas_datareader* is installed the next part of the process is to install *datetime*.[3] *Datetime* is useful for setting dates for retrieving specific data. Given that the data can be accessed daily, the process is as follows:

```
start = datetime.datetime (2019, 1, 1)
end = datetime.datetime (2020, 9, 1)
```

The *start* is the date from which the data begins, and the *end* is where the data ends. Setting the date is important because it allows the different analysis concerning the specific time. With these settings the data can be retrieved by utilizing the code `CPALTT01USM657N` that was used before.

```
consumer_price_index = pdr.DataReader('CPALTT01USM657N', 'fred', start,
end)
```

When the variable is created, the data can be analyzed. To check that the data is correct, the next process is suggested:

```
consumer_price_index.tail()
```

```
df_cpi  = consumer_price_index.rename(columns={'CPALTT01USM657N':
'CPI'})
```

[2] For more information visit: https://pypi.org/project/pip/.

[3] For more information visit: https://docs.python.org/3/library/datetime.html.

DATE	*CPALTT01USM657N*
2020–05-01	0.001950
2020–06-01	0.547205
2020–07-01	0.505824
2020–08-01	0.315321
2020–09-01	0.139275

What can be observed is that the last data in FRED is from the first of May 2020. Given that the code `CPALTT01USM657N` could be difficult for others to understand when sharing the Collaboratory, the title of the column must be changed:

```
df_cpi  = consumer_price_index.rename(columns={'CPALTT01USM657N':
'CPI'})
```

The Gross Domestic Product

When analyzing economic cycles, one of the most important variables is the Gross Domestic Product better known as GDP. The Gross Domestic Product can be measured through different approaches but the most useful is the expenditure approach. The expenditure approach is based on the following formula:

$$GDP = C + I + G + (X - M)$$

C = consumption
I = investment
G = Government expenditure
X = Exports
M = Imports

The approach behind the *GDP* equation is to understand how the economy is funded and from there understand how it works. The theory is based that the households offer funds to the banks as savings or to the companies as investments. Households also offer labor to the companies in returns of funds, salaries, that can be used for consuming the products created by the companies or they can be invested and/or saved. Also, from the salary, the government charges taxes which are used for public goods and services. Finally, exports and imports are based on the exposure of a country to other economies. In this sense, if a country didn´t

share economic activity with other countries there wouldn´t exist exports and imports, just consumption, investment and government expenditure.

In order to understand the economic cycle, the GDP will be analyzed. The Real GDP will be used because it is inflation adjusted and the nominal GDP is established on a base price. The code for the Real GDP is GDPC1). The data for the GDP is in billions of dollars, seasonally adjusted at an annual rate and with a quarterly frequency.

```
gdp = pdr.DataReader('GDPC1', 'fred', start, end)
```

Once the data is retrieved, a change can be made for a more useful analysis. For this change, the *pct.change()*[4] will be used to analyze the growth of the GDP. Using *pct.change()* is simple, when the parenthesis is left blank, the program assumes that there is one period, which is the process that will be used.

```
gdp_change = gdp.pct_change() * 100
```

DATE	*GDPC1*
2019–07-01	0.636915
2019–10-01	0.586232
2020–01-01	–1.262655
2020–04-01	–8.986117
2020–07-01	7.403492

The name of the column can be changed with the process used before.

```
gdp_change = gdp_change.rename(columns={'GDPC1': 'GDP %'})
```

DATE	*GDPC1*
2019–04-01	0.370716
2019–07-01	0.636915
2019–10-01	0.586232
2020–01-01	–1.262655
2020–04-01	–8.986117
2020–07-01	7.478741
2019–04-01	0.370716

[4] For more information please visit: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pct_change.html.

The change in the GDP is important to understand the business cycle. One of the most usual question considering macroeconomics is if the nominal or real growth should be used when analyzing the GDP.

The nominal GDP reflects the growth without the inflation, meaning that it considers the output of production by the country and it analyzes its growth when compared to previous year. Usually, GDP measuring has a base GDP year which establishes the growth for the nominal GDP. The main difference is that when using the real GDP is that it reflects the growth adjusted, leading to the GDP Deflator.

The Gross Domestic Product Price Deflator

The GDP deflator, also known as implicit price deflator for GDP measures the percentual difference between the nominal GDP and the real GDP. It is a useful tool for understanding the change in price because it helps the investor understand if the growth has been motivated by production or by prices. The equation is as follows:

$$GDPDeflator = \frac{nominalGDP}{realGDP} \times 100$$

To calculate the *deflator* of the GDP the *sum* function will be used. The sum function sums all the data points in the list. The first step is to establish a base year, in this case the year will be 2017.

```
start = datetime.datetime (2017, 1, 1)
end = datetime.datetime (2017, 12, 31)
```

```
nominal_GDP = pdr.DataReader(['GDP'],'fred', start, end)
```

The comparison will be the year 2019.

```
start = datetime.datetime (2019, 1, 1)
end = datetime.datetime (2019, 12, 31)
real_GDP = pdr.DataReader(['GDPC1'],'fred', start, end)
```

Once the real GDP and the nominal GDP had been set into a variable, the following process is to create the sum of the production in the year.

```
ngdp_sum = nominal_GDP.sum()
ngdp_sum = int (ngdp_sum)


rgdp_sum =   real_GDP.sum()
rgdp_sum = int (rgdp_sum)
```

Notice that in the process the variables have been converted to integers with the function *int*. The reason for this is that if the data is divided without converting it into an integer, then the problem is that Python interprets the data as two different series and cannot unite them for a process.

With the data converted into integers, then it can be divided.

```
deflator = ngdp_sum/rgdp_sum * 100


102.36361731660686
```

The deflator for 2019 in comparison with the nominal GDP in 2017, the prices have grown by 125%, which reflects that the growth has grown by boost of prices. Considering that investment it is important to have a growth higher than inflation, following the deflator is an important indicator.

Understanding the Process into the Basics

One of the most important process is the use of *sum* instead of the (+) sign seen in the understanding numbers in Python. Basic arithmetic functions can be interchanged with the commands such as *sum*() or *prod*(). These functions are extremely useful when analyzing the data and developing arithmetic operations in data frames.

When understanding the type of numbers Python supports and registers, a number such as the variable *deflator* can be identified with the command *type*. One example as follows:

```
type(deflator)
float
```

In the example above the deflator is a float given that it has decimal numbers. The number could be changed to an integer by using the *int()* function.

```
int(deflator)
102
```

The variables can be changed and therefore it is important to understand what each variable represents. For example, if the variables are developed by a data frame and there is a difficulty when adding them, the variables can be modified to an integer so that they can be summed. An example as follows:

```
deflator = (int (nominal_GDP.sum()))/ (int (real_GDP1.sum())) * 100
```

The process above changes the *sum* of the variable *nominal_GDP* into an integer so that it can be divided into the sum of the integer of the *real_GDP1* variable.

Considering data structures, given that the information will be retrieved from an API during this book the result when analyzing list is a DataFrame.

```
type(real_GDP)
pandas.core.frame.DataFrame
```

As mentioned before, a DataFrame is a library in *pandas* that allow us to create a more integral analysis. The API used during this book converts the data into a DataFrame automatically and there is not a necessity to apply the commands learned before.

As seen in the chapter before, this creates an easy access to the functions and on how to work with them. For a better example, let us use more variables.

Comparing GDP

When analyzing growth, it is important to compare how the growth of the country that is being analyzed compares to the other countries. In this process, a graph will be developed to understand how the different countries behaved.

Installing packages:

```
import pandas_datareader as pdr
import pandas as pd
import datetime
import matplotlib.pyplot as plt
```

Setting a date and retrieving information based on the FRED code:

```
start = datetime.datetime (2015, 1, 1)
end = datetime.datetime (2020, 12, 31)



gdp_comparison = pdr.DataReader(['BRAGDPNQDSMEI',
'CHNGDPNQDSMEI','USAGDPNQDSMEI'],'fred', start, end)
```

Modifying the names on the country list:

```
gdp_comparison =
gdp_comparison.rename(columns={'BRAGDPNQDSMEI':'Brazil','CHNGDPNQDSMEI'
:'China', 'USAGDPNQDSMEI': 'USA'})

gdp_comparison.head()
```

DATE	*Brazil*	*China*	*USA*
2015–01-01	1.488903e+12	1.511379e+13	4.500850e+12
2015–04-01	1.485916e+12	1.685497e+13	4.555894e+12
2015–07-01	1.504503e+12	1.765977e+13	4.586856e+12
2015–10-01	1.516466e+12	1.925729e+13	4.594701e+12
2016–01-01	1.532747e+12	1.624100e+13	4.617539e+12

Changing the data to percentage change:

```
gdp_change = gdp_comparison.pct_change() * 100

gdp_change.head()
```

Dropping NA for creating a chart:

```
gdp_change = gdp_change.dropna()
gdp_change.head()
```

DATE	*Brazil*	*China*	*USA*
2015–04-01	–0.200587	11.520472	1.222980
2015–07-01	1.250849	4.774853	0.679603
2015–10-01	0.795178	9.046097	0.171021
2016–01-01	1.073628	–15.663107	0.497056
2016–04-01	2.034344	11.209285	1.007306

As seen before, this is a DataFrame in which the columns can be edited, the numbers can be dropped and the information changed. This is important when applying certain loops as well as when developing a portfolio, which will be seen in future chapters.

Using Stock Market Data in Python

Abstract The present chapter aims to demonstrate the different access to data concerning securities and the packages that are useful to analyze the data. The book emphasizes Yahoo Finance API, but it explains various API's that are accessible for analyzing the data. The packages are explained and applied to the data.

Keyword API · Packages · Installation · Data

There are different sources of data that are viable in finance. Some of them can be accessed without creating and CVS (comma-separated-values) file or an XLS file (Microsoft Excel). The sources that can be accessed online through Python and without using one of the after mentioned files are the API (application programming interface). The API is useful because since they can have a specific protocol, routines and data structures, the information accessed can be retrieved every time we are using Python.

API Sources

(1) Google Finance: Google developed an API for retrieving information from the financial markets. The API has been discontinued and therefore it is mentioned here as a resource that can be uploaded in the future.

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_4

(2) Yahoo Finance: One of the most important API that the book will be working with, given that it is free and that the information is accurate since it is retrieved directly from the markets. Since Yahoo started its financial platform in 1997 it has grown and its one of the most consulted platforms for financial decisions that is accessible by price and by easiness of language.
(3) Quandl: The company's first movement in financial data was when they launched, in 2013, a million free data set with its universal API. This created the possibility for analyzing data from other sources. In late 2018 they were acquired by Nasdaq (National Association of Securities Dealer), making them one of the most interesting datasets in the markets.

Although there are other interesting API in the market, in the book we will be using the two specific API, Yahoo Finance and Quandl. The reason for using these API is because they are gratuitous and that they are accurate. There are other databases such as Bloomberg that have an API but access to the data could be expensive.

For the use of CVS and XLS databases, the main databases that will be used throughout the book are Yahoo Finance for statistical and portfolio analysis and World Bank and International Monetary Fund for macroeconomic investment strategies.

Most Important Libraries for Using Data in Python in the Present Book

The first API that will be used is the Yahoo Finance API. For this, it is important to know certain packages that will be important in order to handle the different data that will be accessed.

(a) NumPy[1]: The numerical package for Python is one of the most important packages for handling data. By using NumPy the data can be approached for linear algebra, multi-dimensional containers, to create arrays and many other uses. The capability of NumPy will be demonstrated throughout the book.

[1] For more information concerning NumPy: http://www.numpy.org/.

(b) Pandas[2]: The name of the library is an acronym for *Python Data Analysis Library* which is one of the most powerful libraries when it comes to analyzing data. Pandas can be used to create a DataFrame, slicing, replacing, creating time series just to name a few. Pandas will be used throughout the book.
(c) Matplotlib: Throughout the book Matplotlib will be used for plotting 2D graphs. Matplotlib is an excellent library for plotting because of its quality, the variety of graphs that can be elaborated and the easiness of its use.
(d) *f.fn( )*[3]: One of the most interesting libraries for *quantitative finance* is the f.fn( ) library which helps access data and plot easily. In this book, the library will be used to access portfolio and measure performance. It complements with *Matplotlib*[4] when creating graphs.
(e) Ta-lib: Excellent libraries for developing technical analysis and backtesting. Will be used with portfolios and stocks. The installation is tricky, therefore a suggestion is appropriate.

How to install Ta-Lib
The author found it complicated to install Ta-lib. Therefore, it should be run in the *Jupyter Notebook* and it cannot be run in the Google Colab because of the installation. Given that the author utilizes a *Macbook* for programming, the process is only stated for an Apple computer.

First Step: Install Brew in Your Computer.

For installing packages, brew is one of the best in the field. It is useful because it facilitates the process. For downloading brew follow the present link: https://brew.sh/.

After installing brew it is important to download xcode, which will help the process of installation. In this case the installation of xcode it can be done from the Apple Store.

The final process is to install Ta-lib and check that the name of installation has changed to *talib* and it is not as described on the webpage.

[2] For more information concerny Pandas: https://pandas.pydata.org/getpandas.html.
[3] For more information visit: https://pmorissette.github.io/ffn/index.html.
[4] For more information visit: https://ta-lib.org.

To import the other libraries first they must be installed. Please visit the section on Anaconda Navigator and Installing Packages, this will be very helpful in the long run.

Other Important Libraries Not Used in This Book

(a) SciPy[5]: It is a library in Python that uses NumPy and Matplotlib for mathematics and statistics, including signal processing, linear algebra, spatial data and special functions to name a few. For more information

(b) QuantPy[6]: Although the library is in *alpha*, this is one of the libraries that will be interesting to use in the future. The aim is to simplify calculations used in the present book.

(c) TIA[7]: It is described as a *tool kit* to access information, mostly on the Bloomberg® terminal. Given that the *terminal* is not easy to find in many countries, this library is suggested.

(d) PyNance[8]: was one of the most interesting libraries with *f.fn( )* for creating graphs. Sadly part of the code has been deprecated and it has suffered changes which convert it into complicated for use and therefore, it is not recommended for the book.

Suggestion of Libraries for Other Applications

Because the purpose of the book is to analyze stocks and learn to use Python, many of the processes are not created through a specific library, but by following a step-by-step process and then applying it to a specific library. Also, when analyzing a library, the author searched for those that were intuitive for a practitioner moving into Python. The suggestion of libraries below is part of the next steps for the reader which the author found interesting but that the above libraries suffice the implementation with a more accessible framework.

[5] For more information visit: https://docs.scipy.org/doc/scipy/reference/.

[6] For more information visit: https://github.com/jsmidt/QuantPy.

[7] For more information visit: https://github.com/bpsmith/tia.

[8] For more information visit: http://pynance.net.

Trading and Backtesting:

- Trade: visit https://github.com/rochars/trade
- Analyzer: visit https://github.com/llazzaro/analyzer
- Finmarketpy: visit https://github.com/cuemacro/finmarketpy

Options:

- Vollib: Visit https://github.com/vollib/vollib

Risk Analysis:

- Empyrical: Visit https://github.com/quantopian/empyrical
- Qfrm: One of the best libraries for bonds. Visit https://pypi.org/project/qfrm/

Time Series:

- Statsmodels: By far, one of the best libraries for statistical modeling. Visit https://www.statsmodels.org/stable/index.html

Using Python with Yahoo Finance API

The first step for working with Yahoo API is to establish which libraries are going to be used. As it was mentioned earlier, NumPy, Pandas and Matplotlib will be using for accessing the data. The following is the command on Jupyter Notebooks:

```
import NumPy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

The `%matplotlib inline` command is important because it creates a better graph when elaborating a Jupyter Notebook and also the quality is better. It helps the plots to work correctly and it is mentioned as necessary for the use of Matplotlib.

The second step is to import a module called *datetime*[9] and another module called *pandas_datareader*.[10] The *datetime* module is useful for manipulating dates. This is important given that the information regarding Yahoo API must be determined on a given date. Also, it is useful when comparing datasets between different stock quotes.

The *pandas_datareader* allow us to access data from the web. In this case the *pandas_datareader* will allow the usage of the API and of the information.

The script in Jupyter Notebooks is as follows:

```
import pandas_datareader as pdr
import datetime
import pandas_datareader.data as web
```

Once the Jupyter Notebook is set, the Yahoo API can be accessed. The most important aspect is to set a start date and an end date for the values. It is also important to add the stock quote that will be consulted, mainly its ticker. A stock ticker symbol is a one to four letter code representing the name of the company. For this example, the company Tesla will be used for analysis. The ticker for Tesla is TSLA. The information is going to be accessed from January 1, 2015 to January 1, 2019. The code is as follows:

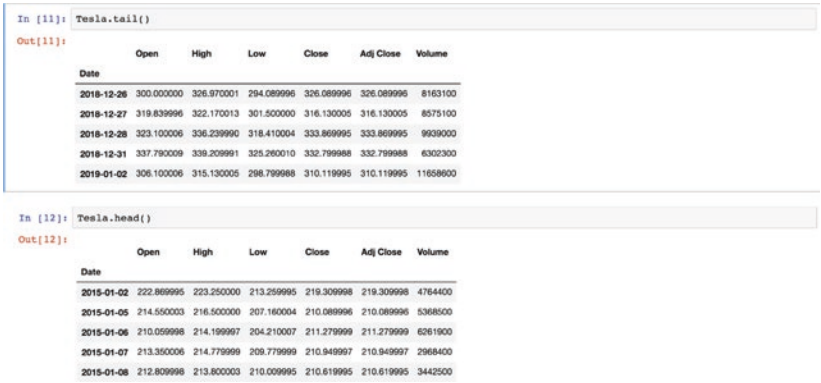
```
start = datetime.datetime(2015,1,1).
end = datetime.datetime(2019,1,1).

Tesla = web.DataReader('TSLA','yahoo',start,end)
```

In the example above two variables were created to define the information that is going to be accessed. The start date is set to January 1, 2015 and the end date is set to January 1, 2019. Both variables are then used in the main variable where the Yahoo API is going to be accessed. The variable Tesla will use the combination of the module *web* with *DataReader* to access the information from the stock company TSLA

[9] For more information regarding datetime module: https://docs.python.org/2/library/datetime.html.

[10] For more information regarding pandas_datareader: https://pandas-datareader.readthedocs.io/en/latest/.



```
In [11]: Tesla.tail()
Out[11]:
```

Date	Open	High	Low	Close	Adj Close	Volume
2018-12-26	300.000000	326.970001	294.089996	326.089996	326.089996	8163100
2018-12-27	319.839996	322.170013	301.500000	316.130005	316.130005	8575100
2018-12-28	323.100006	336.239990	318.410004	333.869995	333.869995	9939000
2018-12-31	337.790009	339.209991	325.260010	332.799988	332.799988	6302300
2019-01-02	306.100006	315.130005	298.799988	310.119995	310.119995	11658600

```
In [12]: Tesla.head()
Out[12]:
```

Date	Open	High	Low	Close	Adj Close	Volume
2015-01-02	222.869995	223.250000	213.259995	219.309998	219.309998	4764400
2015-01-05	214.550003	216.500000	207.160004	210.089996	210.089996	5368500
2015-01-06	210.059998	214.199997	204.210007	211.279999	211.279999	6261900
2015-01-07	213.350006	214.779999	209.779999	210.949997	210.949997	2968400
2015-01-08	212.809998	213.800003	210.009995	210.619995	210.619995	3442500

Fig. 1 Example of the retrieval of data from Tesla (*Source* Obtained from the computer of the author)

from the API Yahoo in the dates from January 1, 2015 to January 1, 2019.

Now the information is in the current Jupyter Notebook. It is important to recall that every time that we shut down the Jupyter Notebook the data must be uploaded again using the same procedure or by running the whole script. To visualize the information the following command can be executed:

Tesla.tail( ) or Tesla.head( )

The.tail command will show the last five dates of the data. The.head command will show the first five dates in the beginning of the dates that were established. Figure 1 demonstrates the capacity of the Yahoo API.

From the Yahoo API the Opening price, the Highest Price of the day the Lowest Price of the day the Adjusted Close and the Volume are displayed.

Using Python with Quandl API

Quandl is an excellent source of information because it offers databases for both free and premium. The first step for using Quandl is creating a user at the signup page at https://www.quandl.com/. This is an important aspect because Quandl offers an API Key that is necessary for accessing different data.

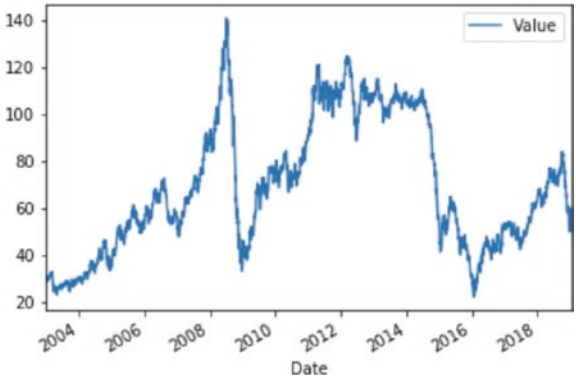


Fig. 2 Petroleum Prices using Quandl (*Source* Elaborated by the author with information from Quandl)

Once the user is created in Quandl, in the account settings it will appear *Your API Key* with a lookalike to the following *c2_teStKey*. The key is personal and it should not be given to other users for access.

With the key, the script in Jupyter is quite easy. The first step will be to import Quandl, the second step will be to import the API Key and then create a search based on the Petroleum Prices reported by the Organization of the Petroleum Exporting Countries (OPEE) from the following Quandl address: https://www.quandl.com/data/OPEC/ORB-OPEC-Crude-Oil-Price. The script is as follows:

```
import quandl
quandl.api_config.api_key = 'c2_TeStKey'
petroleum = quandl.get("OPEC/ORB", authtoken="c2_xdULhUk6HxnA1kfja")
```

Depending on the database there are different aspects that could be used. For example, the use of the API Key could be inside the variable because of the need of the program. Another important information regarding Quandl is that there are databases that are viable when using a free account such as the OPEC database. An example of the database is as follows, with the script mentioned above (Fig. 2):

```
import matplotlib as plt
%matplotlib inline
petroleum.plot();
```

Using f.fn( ) for Retreiving Information

One of the most important libraries for retrieving information that we will be using in the present book is *f.fff( )*.[11] Certain processes throughout the book are simplified by the use of this package, the most important difficulty is interpreting the results. Therefore, this book will center on developing a traditional approach and then applying certain packages for the retrieval of information.

The first step is to install f.fn( ) in the computer. For this exercise Google Colab will be used:

```
pip install ffn.
```

After the installation it is important to *import* certain packages that *f.fn ( )* uses. The recommended packages are as follows:

```
import ffn
import datetime
import pandas_datareader as pdr
import matplotlib.pyplot as plt
%matplotlib inline
```

Now that it has been installed, the process for retrieving information for Yahoo Finance is as follows:

```
## adding stocks
stocks = ffn.get('aapl, spy, amzn, fb', start='2019-01-01', end='2021-01-
01')
```

As the example above states, the dates are included when getting the stocks. The stocks are separated by a comma (,) and in the order they are selected is the order that the stocks will be reflected in the DataFrame.

```
stocks.tail()
```

[11] For more information please visit: https://pmorissette.github.io/ffn/ffn.html#ffn.core.calc_mean_var_weights.

	aapl	*spy*	*amzn*	*fb*
Date				
2020-12-24	131.970001	369.000000	3172.689941	267.399994
2020-12-28	136.690002	372.170013	3283.959961	277.000000
2020-12-29	134.869995	371.459991	3322.000000	276.779999
2020-12-30	133.720001	371.989990	3285.850098	271.869995
2020-12-31	132.690002	373.880005	3256.929932	273.160004

An important aspect is that *f.fn( )* uses adjusted close as a default. If the interest is to access different prices the process is as follow:

Closing prices:

```
stocks = ffn.get('aapl:Close, spy:Close, amzn:Close, fb:Close',
start='2019-01-01', end='2021-01-01')
```

```
stocks.tail()
```

	aaplclose	*spyclose*	*amznclose*	*fbclose*
Date				
2020-12-24	131.970001	369.000000	3172.689941	267.399994
2020-12-28	136.690002	372.170013	3283.959961	277.000000
2020-12-29	134.869995	371.459991	3322.000000	276.779999
2020-12-30	133.720001	371.989990	3285.850098	271.869995
2020-12-31	132.690002	373.880005	3256.929932	273.160004

High prices:

```
stocks = ffn.get('aapl:High, spy:High, amzn:High, fb:High', start='2019-
01-01', end='2021-01-01')
```

```
stocks.tail()
```

	aaplhigh	*spyhigh*	*amznhigh*	*fbhigh*
Date				
2020-12-24	133.460007	369.029999	3202.000000	270.399994
2020-12-28	137.339996	372.589996	3304.000000	277.299988
2020-12-29	138.789993	374.000000	3350.649902	280.510010
2020-12-30	135.990005	373.100006	3342.100098	278.079987
2020-12-31	134.740005	374.660004	3282.919922	277.089996

Low prices:

```
stocks = ffn.get('aapl:Low, spy:Low, amzn:Low, fb:Low', start='2019-01-
01', end='2021-01-01')
```

```
stocks.tail()
```

	aapllow	*spylow*	*amznlow*	*fblow*
Date				
2020-12-24	131.100006	367.450012	3169.000000	266.200012
2020-12-28	133.509995	371.070007	3172.689941	265.660004
2020-12-29	134.339996	370.829987	3281.219971	276.279999
2020-12-30	133.399994	371.570007	3282.469971	271.709991
2020-12-31	131.720001	371.230011	3241.199951	269.809998

Using Python with Excel

When using Excel in Python and after the installation of Python using Anaconda, it is important to know which directory Python is working on. The following script is necessary for getting to know which directory Python is working and also on knowing where your files are located.

Pdw

```
'/Users/mauriciogarita'
```

In the above example the directory of the Jupyter Notebook is shown. The file that is going to be used has to be in the same directory as the Python in Jupyter Notebook. This is one of the easiest ways of retrieving the information of an excel. The other option is to know where the documentation is.

If the documentation is on the same folder or directory as the Jupyter Notebook that is being used, then the script is as follows:

import pandas as pd.

df = pd.read_excel (r'/Users/mauriciogarita/Desktop/example.xlsx')

df

	a	b	c	d	e
0	2	1	4	3	2
1	3	4	1	2	3
2	4	3	1	2	1
3	5	2	4	3	5
5	5	6	5	3	2
6	6	5	2	1	2
7	4	3	3	4	8
8	3	2	8	7	2
9	4	1	4	2	4
10	5	2	4	2	9

There are other commands such as *pd.read_csv*[12] that can be used when the file is comma-separated (CSV), or *pd.read_table*[13] that can be used for a DataFrame that is delimited. Normally, the easiest version of the *pd.read* is the *pd.read_excel*[14] because it reads the table, even if it is not delimited and turns it into a DataFrame that is extremely useful when handling data.

[12] More information concerning the pd.read_csv can be accessed at: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html.

[13] More information concerning the pd.read_table can be accessed at:https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_table.html.

[14] More information concerning the pd.read_excel can be accessed at: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html.

Conclusion Regarding Using Data in Python

Python has become a popular language because of the easiness it has when it comes to handling data. There are different databases that can be used through an API which includes the information from Quandl, Google, Yahoo Finance, IEX Finance, Alpha Vantage to name a few. Using API is the quickest way to access data and to gather different information depending on the interest on who is accessing the data. It is also the fastest way to refresh a model and get it up to date.

Of course, that not all the information has an API, for example in a company that is private. The use of Excel documents is also viable and fast in Python. As shown in the examples it is easy to access the data from a file, even if it is a CSV or a table. In the next chapter the information will be combined with financial statistics to access data and to create financial models.

Statistical Methods Using Python for Analyzing Stocks

Abstract Statistical Methods are part of the tools for analyzing securities. The following chapter explains the central limit theorem, returns, ranges, boxplots, histograms and other sets of statistical measures for the analysis of securities using Yahoo Finance API.

Keywords Central limit theorem · Returns · Plots · Statistical measures

This next part of the book is centered on the use of mathematical and statistical methods to understand the security based on quantitative analysis. The aim of quantitative analysis is to extract a value that explains financial behavior (Keaton 2019).

The Central Limit Theorem

The Central Limit Theorem (CLT) is part of the study concerning probability theory which states that if random samples of a certain size (n) from any population, the sample will approach a normal distribution. A normal distribution happens when there is no left or right bias in the data (Ganti 2019).

The usual representation of a normal distribution is the *Bell Curve* which looks lithe a bell, hence the name. CLT establishes that given a

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_5

sufficiently large sample size from a population with a specific amount of variance that is finite, the mean is equal to the median and equal to the mode. The meaning of this is that there is complete symmetry in the data and that 50% of the values are higher than the mean and 50% are lower than the mean.

The **mean** is usually divided into two: (1) population mean and (2) sample mean. These aspects are important when analyzing the data that is going to be used. The formulas give information regarding these aspects:

Equation 1: Population mean

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \tag{1}$$

Equation 2: Sample mean

$$\bar{x} = \frac{\sum_{i}^{n} x_i}{n} \tag{2}$$

N = number of items in the population
n = number of items in the sample

The formulas are basically the same with one important difference, that the population mean centers on the number of items of a population and the sample mean on a specific sample. The population should be seen as the total observations that can be made and the sample is usually a part of the population that is selected.

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader
import datetime
import pandas_datareader.data as web
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Setting the continuos data range*

```
start = datetime.datetime(2015,1,1)
end = datetime.datetime(2019,1,1)
```

- *Creating the variable*

```
IBM = web.DataReader(IBM,'yahoo',start,end)
```

Given that the information used for the IBM security is from January 1st, 2015 to January 1, 2019, this is considered as a sample. Therefore, the $\bar{x}$ will be considered as the mean of the security. To calculate the mean, as the equation suggests, it is the sum of the elements divided by the count of the elements.

```
IBM['Close'].mean()
```

- Result

```
151.81089374875862
```

The other aspect mentioned on the CLT is the **median**. The median is the middle value of the set of numbers. To calculate the median in Python it should be calculated as follows:

```
IBM['Close'].median()
```

- Result

```
152.5
```

Which one to choose? The *rule of thumb* is to use the mean when there are no outliers and to use the median when there are outliers.

The third measure of the CLT is the **mode**. The mode is the most frequent point of data in our data set. In a histogram, it is the highest bar. To calculate it in Python:

```
IBM['Close'].mode()
```

- Result

```
146.479996
```

Fig. 1 IBM results using describe

	ibmclose
count	1006.000000
mean	151.847276
std	13.381614
min	107.570000
25%	144.869999
50%	152.504997
75%	160.372505
max	181.949997

The process can be combined with the function *describe* and the package *f.fn ()* for obtaining certain statistical measures of Central Tendency. The process is as follows (Fig. 1):

```
stocks = ffn.get('ibm:Close', start='2015-01-01', end='2019-01-01')

stocks.describe()
```

Creating a Histogram

One of the most important plots to display information in finance is a histogram. A histogram demonstrates the frequency of data that is continuous. The meaning of statistical frequency is the times that a value

occurs on a set of data. Continuous, refers to continuous data, which means that the data can take any value within a range.

The histogram for analyzing security has to be elaborated with the total security return. To calculate the total stock return, the following formula will be used:

Equation 3: Total security return

$$Total\ \text{Security}\ \text{Return} = \frac{(P_1 - P_0)}{P_0} \tag{3}$$

P_1 = Actual Price
P_0 = Previous Price

For example, the company IBM will be used from the before example. Once the variable *IBM* is created there are different approaches to which prices should be used to calculate returns. The conventional approach is to use closing prices, since the prices are registered as the last price on the stock. This is useful when working on a historical database.

If there is a necessity for using the actual data, the recommendation is to use the opening price as the first day and the closing price as the last day. There is also a discussion concerning the adjusted closing price, but this will be seen in later chapters.

- *Creating the returns*

```
IBM_returns = IBM['Close'].pct_change(1)
```

The *IBM_returns* reflects the change in an IBM price. The *pct_change* is part of Pandas that gives the percentage of change between the current and prior number. By establishing the number one (1) it compares the actual price with the price before.

The *IBM_returns* can now be converted into a histogram. For this the *matplotlib.pyplot.hist* will be used. One of the most important attributes of *hist* is the number of bins. The number of bins has as a *Rule of Thumb the* Sturge's Rule.

Equation 4: Sturges rule

$$K = 1 + 3.322\ log_N \tag{4}$$

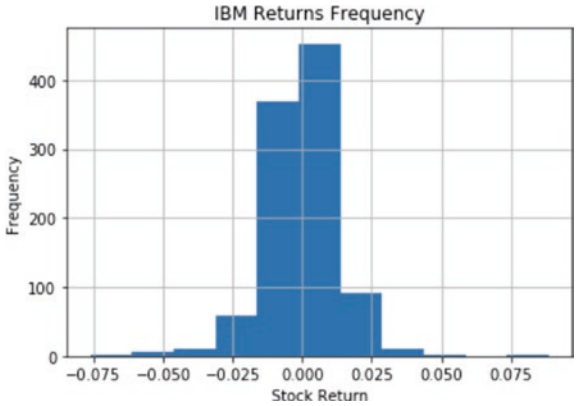


Fig. 2 IBM Returns Frequency (*Source* Elaborated by the author with information from Yahoo Finance)

K = number of bins
N = number of observations in the sets

To obtain the number of observations the Pandas *describe* function can be used as a very useful tool.

IBM_returns.describe()

- Result

```
count    1006.000000
mean       -0.000254
std         0.013023
min        -0.076282
25%        -0.006450
50%         0.000220
75%         0.006213
max         0.088645
Name: Close, dtype: float64
```

The total observations are 1006. By including this value, Sturge's Rule can be calculated.

```
bins = 1+3.3222*np.log10(1006)
```

- Result

```
10.975231011547681
```

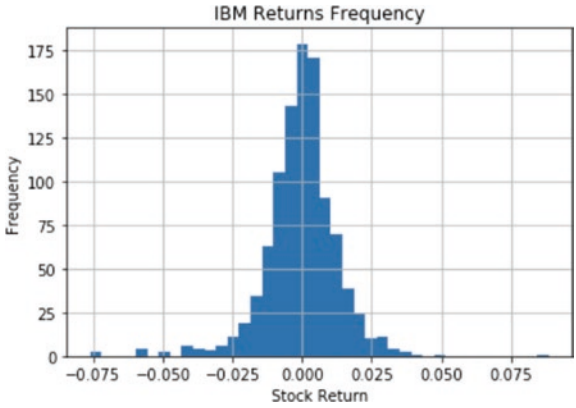


Fig. 3 IBM histogram with 40 bins (*Source* Elaborated by the author with information from Yahoo Finance)

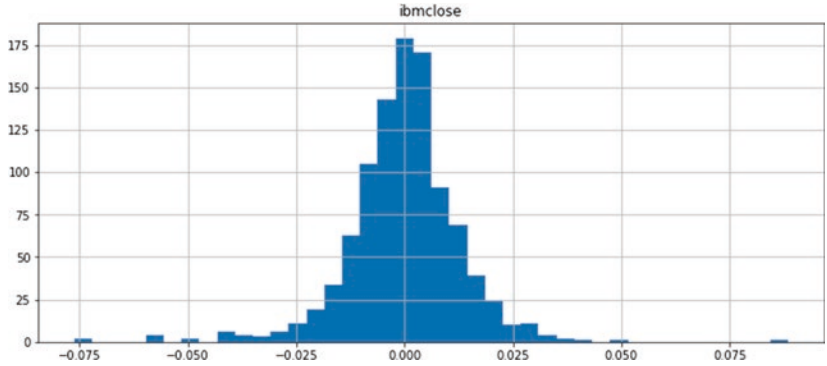


Fig. 4 IBM frequency using f.fn()

The recommended bins according to Sturge's Rule is 11. Now the histogram may be created using 11 bins (Fig. 2).

```
IBM_returns.hist(bins=11);
```

It is important to remember that Sturge's Rule is a *rule of thumb*. If the result does not satisfy the criteria, this can be adapted (Fig. 3).

```
IBM_returns.hist(bins=40);
```

The same process can be followed with the *f.fn()* package using a simpler process as follows (Fig. 4):

Calculate percentage returns:

```
returns = stocks.to_returns().dropna()
```

Creating a histogram:

```
returns.hist(bins=40, figsize=(12, 5));
```

There is another approach for calculating returns and creating a histogram. This is through the logarithmic returns.[1] The reason for using logarithmic returns is the normalization of the data, meaning that the comparison of the data is achievable given that the logarithmic returns convert the data into a more equal series (Brooks 2008). This assumes that the distribution is normal, an important aspect of descriptive statistics.

Equation 5: Logarithmic return equation

$$Logarithmic\ return = ln\left(\frac{P_1}{P_0}\right) \tag{5}$$

$ln =$ Natural Logarithm
$P_1 =$ Actual Price
$P_0 =$ Previous Price

To achieve this in Python, it is necessary to use the *shift* and the *np.log*. The *shift*[2] is used to move the calculation by one value, hence the name. The *np.log*[3] returns the natural logarithm with a base e. Both of them are necessary to use in order to calculate the logarithmic returns.

```
IBM_log_returns = np.log(IBM['Close']/IBM['Close'].shift(1))
```

[1] A recommended article on the subject: https://quantivity.wordpress.com/2011/02/21/why-log-returns/.

[2] Pandas documentation on shift: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.shift.html.

[3] Numpy documentation on np.log: https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.log.html.

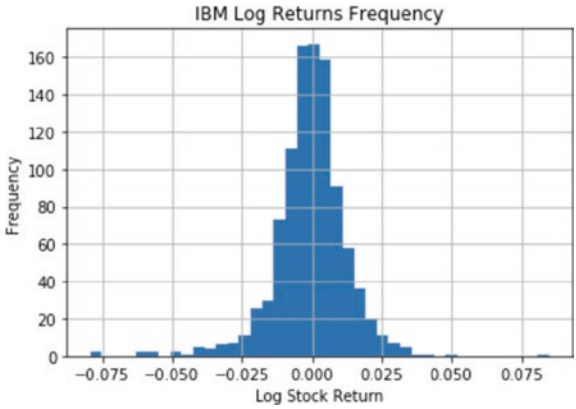


Fig. 5 IBM histogram with logarithmic returns (*Source* Elaborated by the author with information from Yahoo Finance)

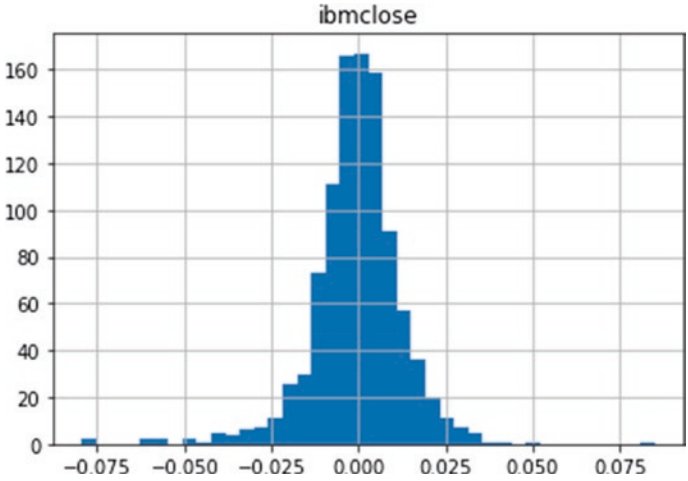


Fig. 6 IBM frequency using f.fn() with logarithmic returns

When elaborating the comparison of the data, the change between percentage or logarithmic return is not noticeable, because it is usually in the fifth decimal point (Fig. 5).

The process can also be elaborated with the *f.fn()* package by calculating the logarithmic returns as follows (Fig. 6):

```
returns = stocks.to_log_returns().dropna()
```

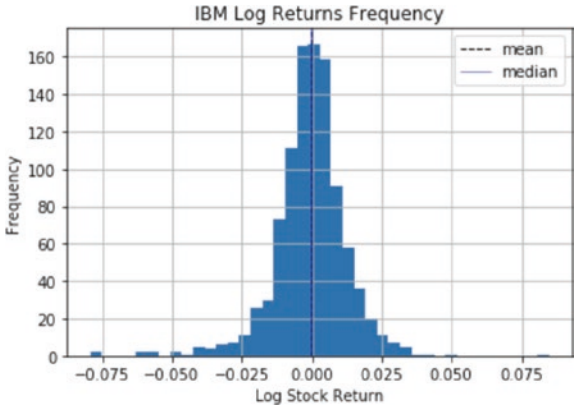


Fig. 7 IBM Returns with axvline (*Source* Elaborated by the author with information from Yahoo Finance)

After calculating the returns, the process of creating the histogram is similar:

```
returns.hist(bins=40, figsize=(12, 5));
```

Creating a Histogram with Line Plots

The above results can be charted into the histogram and the line plot. To add the results into the histogram an *axvline* can be added as a part of the plot. The *axvline* allows a line to be created to define the plot. In the script, the mean and the median of the logarithmic returns have been added (Fig. 7).

```
IBM_log_returns.hist(bins=40);
_ = plt.xlabel('Log Stock Return')

_ = plt.ylabel('Frequency')

_ = plt.title('IBM Log Returns Frequency')

_ = plt.axvline(IBM_log_returns.mean(), color='k', linestyle='dashed', linewidth=1, label =
'mean')

_ = plt.axvline(IBM_log_returns.mean(), color='b', linewidth=0.5, 'median')

plt.legend();
```

The *plt.axvline* was created with the mean function and with the median function. The color[4] was modified and a label was added so that the legend becomes useful when analyzing the data. Given that the mean and the median are very similar, there is almost no difference in the graph but in extreme cases the difference between the mean and the median can be considerable.

For this, it is necessary to discuss about **skewness**.[5] Since it is important to understand symmetry, skewness is useful as a measure for understanding the lack of symmetry. When talking about skewness, the information can be skewed to the left, to the right or be at the center point (Jain 2018).

IBM_log_returns.skew()

- Result

```
−0.65950468511581717
```

The skewness for a normal distribution should be zero and the symmetric data should be near this number.

- If the values are positive: data is skewed to the right
- If the values are negative: data is skewed to the left

In the example of IBM, it can be concluded that the data is skewed to the left and that is near zero which leads to being considered symmetric.

Conjointly with measuring skewness it is important to measure **kurtosis**.[6] Kurtosis is the measure of the tails in a normal distribution (Kenton 2019). A high kurtosis is related to having heavy tails, which means outliers. A low kurtosis means lack of outliers which is light tails. It is uncommon to have a uniform distribution. To obtain a kurtosis in Python, the command should be as follows:

IBM_log_returns.kurtosis()

- Result

```
6.2497976901878483
```

[4] The different colors can be consulted at: https://matplotlib.org/examples/color/named_colors.html.

[5] Pandas information for skewness: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.skew.html.

[6] For more information on kurtosis in Pandas: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.kurtosis.html.

There are three options of kurtosis for interpreting the result.

- Leptokurtic: the value of kurtosis is greater than (>) than zero. The interpretation is that the data is centered around the mean.
- Mesokurtic: the value of the kurtosis is equal (=) to zero. This represents a normal distribution
- Platykurtic: the value of the kurtosis is less than (<) than zero. The interpretation is that the data is far from the mean.

In the example of IBM, the kurtosis is leptokurtic, since the values are around the mean and the shape of the *bell* is taller in its area.

Histograms Using f.fn()

To create a histogram with one variable with the use of f.fn() the process is as follows:

Histogram (Percent Change) with Two Variables

The next process is to analyze return by creating a histogram that can be compared between two companies.

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Set the dates for the analysis*

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

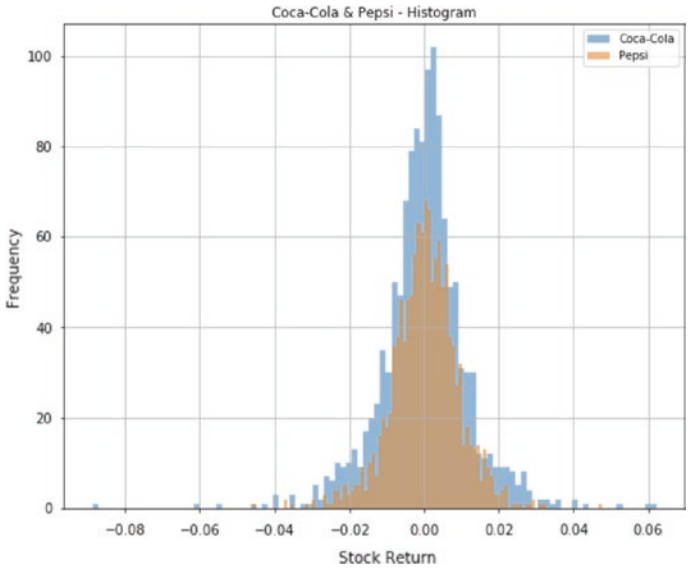


Fig. 8 Coca-Cola and Pepsi percent change Histogram (*Source* Elaborated by the author with information from Yahoo Finance)

- *Select the companies*

```
CocaCola = web.DataReader('K', 'yahoo', start, end)
Pepsi = web.DataReader('PEP', 'yahoo', start, end)
```

- *Percent Change*

```
Pepsi['Returns'] = Pepsi['Close'].pct_change(1)
CocaCola['Returns'] = CocaCola['Close'].pct_change(1)
```

One of the columns are created by the append method. The histogram for both companies can be elaborated as follows (Fig. 8):

- *Creating a histogram*

```
CocaCola['Returns'].hist(bins=100,label='Coca-Cola',alpha=0.5)
Pepsi['Returns'].hist(bins=100,label='Pepsi',figsize=(10,8),alpha=0.5)
plt.legend();
```

Histogram (Logarithmic Return) with Two Variables

The process for creating a comparison with two companies concerning logarithmic returns is the same as the process for one company. To create logarithmic returns using the append method, it can be done as follows (Fig. 9):

- *Calculating logarithmic returns*

```
Pepsi['LN Returns'] = np.log(Pepsi['Close']/Pepsi['Close'].shift(1))
CocaCola['LN Returns'] = np.log(CocaCola['Close']/CocaCola['Close'].shift(1))
```

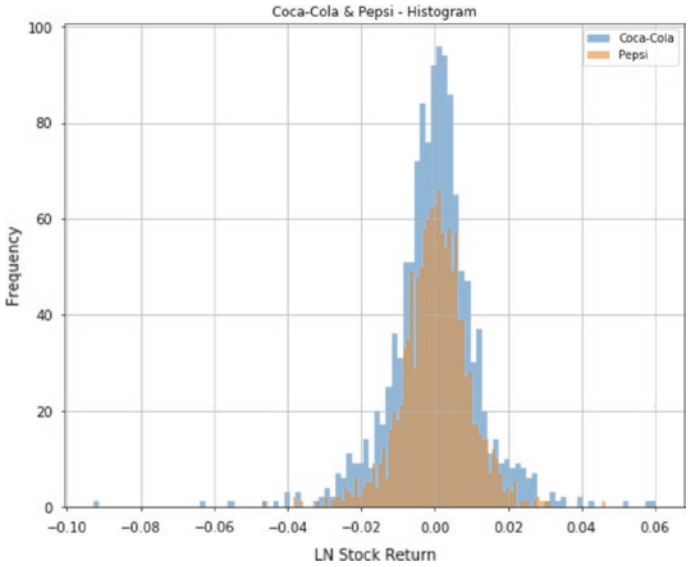


Fig. 9 Coca-Cola and Pepsi logarithmic histogram (*Source* Elaborated by the author with information from Yahoo Finance)

- *Creating a histogram*

```
CocaCola['LN Returns'].hist(bins=100,label='Coca-Cola',alpha=0.5)
Pepsi['LN Returns'].hist(bins=100,label='Pepsi',figsize=(10,8),alpha=0.5)

_ = plt.xlabel('LN Stock Return')

_ = plt.ylabel('Frequency')

_ = plt.title('Coca-Cola & Pepsi - Histogram')

plt.legend();
```

Interquartile Range and Boxplots

In finance, it is extremely important to measure the spread of data. The first aspect for measuring the spread of data is the **range.** The range, as seen before, is the difference between the highest and lowest values (Kalla 2020). This is important because it helps in understanding where the values of the data are located. For calculating the range, the *max* and *min* functions can be used.

```
IBM_close = IBM['Close']

range_returns = IBM_close.max() - IBM_close.min()
range_returns
```

- Result

```
74.379997000000003
```

Since the max value of the series is 181.94 and the minimum value is 107.57, the difference is the range. This is important to know where the data is located, between 181.94 and 107.75.

Another important range measure is the **Interquartile range (IQR)**. The interquartile range is the distance between the third quartile (Q3)

and the first quartile (Q1) (Wan et al. 2014). The Q1 should be understood as one-fourth (25%) of the data is the same or less than the Q1 result. The Q3 should be understood as three-fourths (75%) of the data is the same or the Q3 result.

The measure is important for understanding outliers. The *rule of thumb* is that if a value is larger than Q3 plus 1.5 times the IQR, then this value is an outlier. This also applies if the value is Q1 minus 1.5 times the IQR range.

Equation 6: IQR formula

$$IQR = Q_3 - Q_1 \tag{6}$$

Outliers

$$If\ datapoint > Q_3 + 1.5 \times IQR$$

$$If\ datapoint < Q_1 - 1.5 \times IQR$$

In Python the IQR can be calculated as follows:

```
IBM_Q1 = IBM_close.quantile(0.25)
IBM_Q1
```

- Result

```
144.84000400000002
```

```
IBM_Q3 = IBM_close.quantile(0.75)
IBM_Q3
```

- Result

```
160.3650055
```

```
IBM_IQR = IBM_Q3 - IBM_Q1
IBM_IQR
```

- Result

```
15.525001499999973
```

For the outliers, the process is as follows:

```
IBM_outlier_high = IBM_Q3 + 1.5 * IBM_IQR
IBM_outlier_high
```

- Result

```
183.65250774999996
```

```
IBM_outlier_low = IBM_Q1 - 1.5 * IBM_IQR
IBM_outlier_low
```

- Result

```
121.55250175000006
```

Given the result, there are outliers in the lower range, given that the minimum value of the is 74.38 and the low outlier is 121.55. There is an intense discussion about removing outliers, or retaining them considering that outliers are useful. For the present moment, the outliers should be kept.

The visualization for analyzing outliers is the boxplots. The boxplots are also known as the box-whisker plots and they are useful to understand the concentration of the data. To create a boxplot the information needed is as follows:

- Minimum value outlier
- Q1
- Median
- Q3
- Max value outlier

To create a boxplot in Python (Fig. 10):

```
plt.boxplot(IBM_close);
_ = plt.ylabel('IBM Closing Price')
_ = plt.title('IBM Close - Boxplot')
_ = plt.xlabel('IBM')
```

As the boxplot demonstrates, the highest value for measuring an outlier is 183.65. The beginning of the box below the max value is Q3 (160.35). The orange line in the middle of the box is the median (152.5). Below is the Q1 range (144.84) and the last line is the smallest value for measuring an outlier (121.55). The white and black points are the outlier of the series, just as calculated by the IQR range and the *rule of thumb* for outliers.

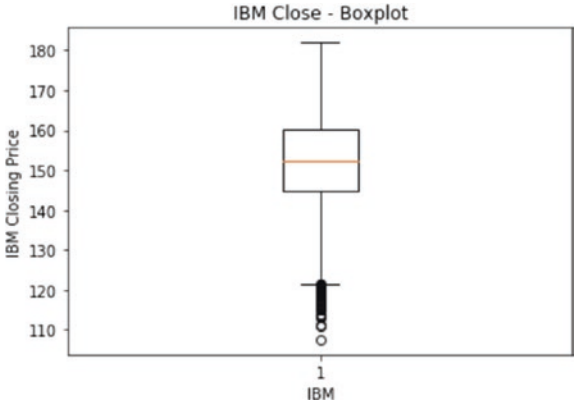


Fig. 10 IBM Boxplot (*Source* Elaborated by the author with information from Yahoo Finance)

Boxplot with Two Variables

The creation of a **Box-Plot** is certainly complicated. For instance, both variables, Pepsi['Close'] and CocaCola['Close'], have to be included in the same variable. The reason for uniting both variables is that if the process is not done with concatenation, the boxplot will be graphed as one above the other. To concatenate the *pd.concat* function of Pandas is extremely useful (Fig. 11).

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

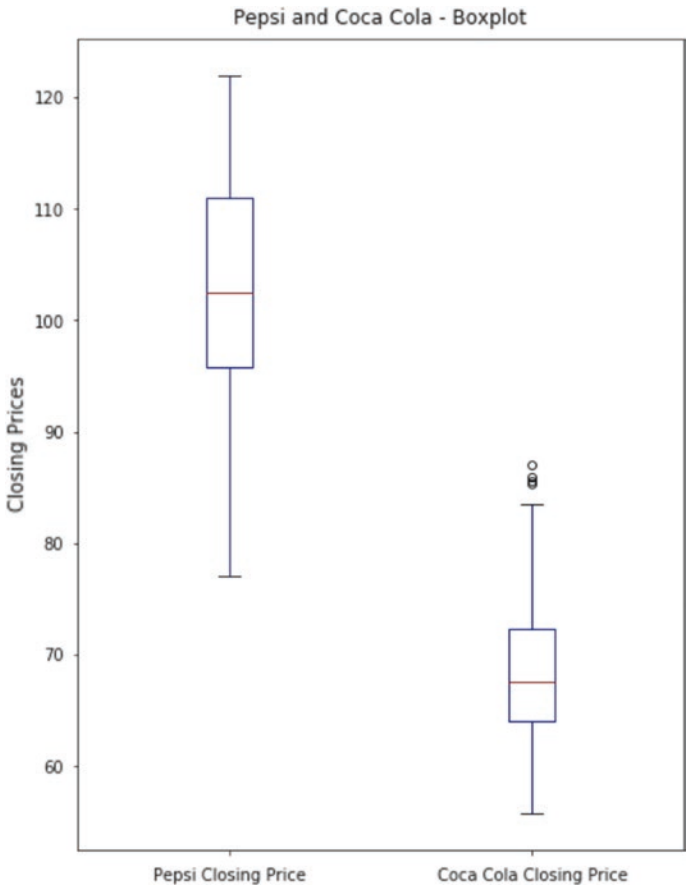


Fig. 11 Coca-Cola and Pepsi box-plots (*Source* Elaborated by the author with information from Yahoo Finance)

- *Set the dates for the analysis*

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

- *Select the companies*

```
CocaCola = web.DataReader('K', 'yahoo', start, end)
Pepsi = web.DataReader('PEP', 'yahoo', start, end)
```

- *Concatenating*

```
united_box = pd.concat([Pepsi['Close'],CocaCola['Close']],axis=1)
```

The next step is to create the columns in order to identify clearly the difference between the prices. The process is as follows:

```
united_box.columns = ['Pepsi Closing Price','Coca Cola Closing Price']
```

For plotting, getting the information elaborated with the columns and combining them for graphs.

```
united_box.plot(kind='box',figsize=(8,11),colormap='jet')


_ = plt.ylabel('Closing Prices')


_ = plt.title('Pepsi and Coca Cola - Boxplot')
```

The boxplot demonstrates the outliers concerning Coca-Cola and Pepsi, as well as the range of the prices. Considering that Pepsi's closing prices are higher, this creates an important difference between the height of the graphs. It is important to consider that the information selected can be applied to more than two variables.

Kernel Density Plot and Volatility

Although the IQR range is important, there are other elements that measure the spread of data that are vital in finance: (1) variation and (2) standard deviation. **Variance** measures the spread between the numbers of the data set (Hayes 2019). The formula is as follows:

Equation 7: Variance formula

$$\sigma^2 = \sum \frac{(x - \bar{x})}{n} \tag{7}$$

x = data point
$\bar{x}$ = mean of the data points in the series
n = total of data points

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader
import datetime
import pandas_datareader.data as web
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Setting the continuos data range*

```
start = datetime.datetime(2015,1,1)
end = datetime.datetime(2019,1,1)
```

- *Creating the variable*

```
IBM = web.DataReader(IBM,'yahoo',start,end)
```

- *Calculating the variance*

IBM_log_returns.var( )

- Result

```
0.00017102549386496948
```

The interpretation of the number should be based on a large or small variance. In finance, a high variance can be led to a riskier asset, meanwhile, a low variance should be interpreted as a low-risk asset. For a more accurate analysis the **standard deviation** is used.

Equation 8: Standard deviation equation

$$\sigma = \sqrt{\sum \frac{|(x - \bar{x})|}{n}} \tag{8}$$

x = data point
$\bar{x}$ = mean of the data points in the series
n = total of data points

The standard deviation measures the dispersion of the data set to its mean. The standard deviation in finance is often referred to as **volatility** (Hargrave 2020). Volatility is important because when analyzing higher volatility, it is usually related to risk. In python it should be computed as follows:

IBM_log_returns.std()

- Results

```
0.0130776715765831
```

Since the standard deviation cannot be negative (because it uses absolute values) and the smallest standard deviation possible is zero, the standard deviation of IBM logarithmic returns can be seen as a small variation and therefore, a less risky asset. For this it is important to compare IBM to other companies.

There is a discussion concerning standard deviation, variance and IQR range. The main comment is that standard deviation and variance are affected by outliers and therefore, it can be inaccurate. Since IQR range considers the identification of assets, and it is sometimes seen as a better measure. Even so, standard deviation and variance are commonly used in finance.

Finally, one last statistical tool is the Kernel Density Estimation better known as the KDE. The KDE plot is usually a replacement for the histogram because it is calculated by applying a weight to the distance between points in the dataset. The weighing of distance is achieved by the following formula:

Equation 9: Kernel density estimation—weighting

$$\textit{Weighting distance} = \sum K\left(\frac{x - \textit{observation}}{\textit{bandwith}}\right) \tag{9}$$

K = The kernel function
Bandwidth: distance between observation
x = observations

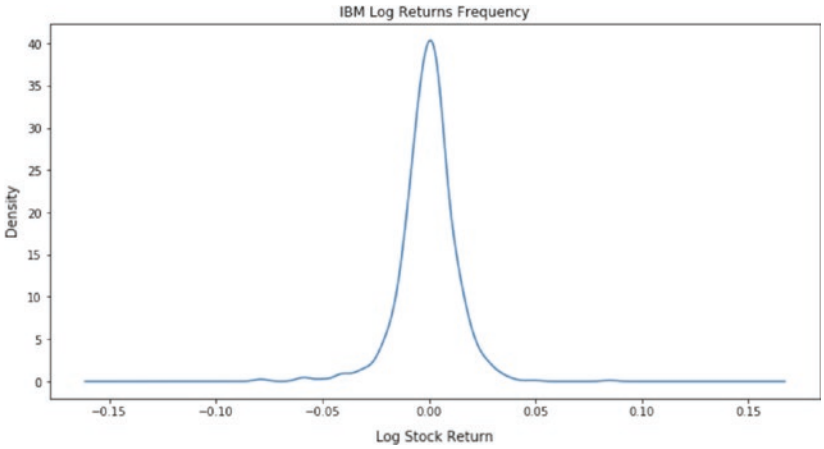


Fig. 12 IBM KDE with log returns (*Source* Elaborated by the author with information from Yahoo Finance)

The Kernel function can be an Epanechnikov, normal, uniform or triangular. The Pandas data frame *kde* uses the *scott rule*[7] which is equivalent to the proposal of D.W Scott which establishes an approach to a normal density.

In conclusion, a KDE is a smoother version of the histogram and can be compared easily. It is usually graphed with the histogram to compare the behavior of the variable. In the exercise below it has been plotted individually (Fig. 12):

```
IBM_log_returns = np.log(IBM['Close']/IBM['Close'].shift(1))

IBM_log_returns.plot(kind='kde',bw_method='scott', label='IBM',figsize=(12,6));

_ = plt.xlabel('Log Stock Return')

_ = plt.ylabel('Density')

_ = plt.title('IBM Log Returns Frequency')
```

[7] For more information concerning the *scott rule*: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html#scipy.stats.gaussian_kde.

Kernel Density Plot (Percent Change) with Two Variables

The elaboration of the KDE plot is simple once the returns have been created. For this reason, the process is as follows (Fig. 13):

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Set the dates for the analysis*

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

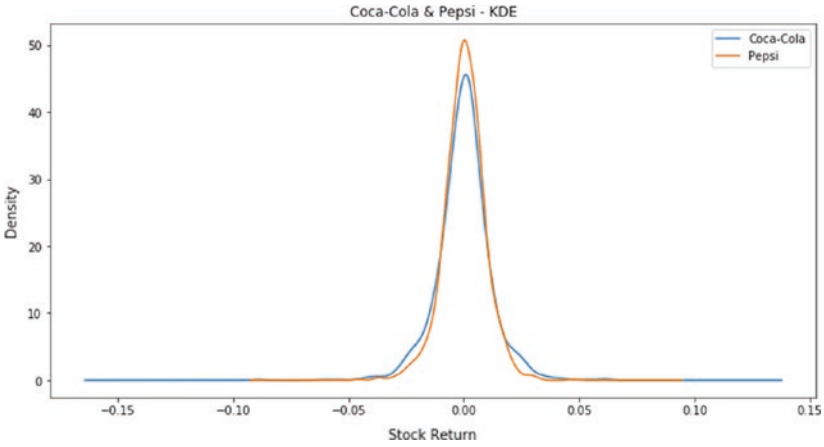


Fig. 13 Coca-Cola and Pepsi KDE (*Source* Elaborated by the author with information from Yahoo Finance)

- *Select the companies*

```
CocaCola = web.DataReader('K', 'yahoo', start, end)
Pepsi = web.DataReader('PEP', 'yahoo', start, end)
```

- *Calculating returns*

```
Pepsi['Returns'] = Pepsi['Close'].pct_change(1)
CocaCola['Returns'] = CocaCola['Close'].pct_change(1)
```

- *Ellaborating the KDE*

```
CocaCola['Returns'].plot(kind='kde',label='Coca-Cola',figsize=(12,6))
Pepsi['Returns'].plot(kind='kde',label='Pepsi')

_ = plt.xlabel('Stock Return')

_ = plt.ylabel('Density')

_ = plt.title('Coca-Cola & Pepsi - Histogram')

plt.legend();
```

Covariance and Correlation

When comparing stocks, the technical aspects are important to understand the behavior of the stock when trying to build a portfolio, but they can be subjective and often misunderstood. There is an interesting discussion concerning technical analysis and other types of financial analysis such as fundamental or econometric. It is important to understand the tool one is using to make the most out of it.

When trying to understand the relation between different asses there are two quantitative approaches that are very useful: covariance and correlation. Covariance is useful when comparing how two assets are related (Hayes 2019). It is also an important part of the Capital Asset Pricing Model (CAPM) that will be explained in future chapters. The equation of the covariance is as follows:

Equation 10: Covariance

$$Cov(R_a, R_b) = E\{[R_a - E(R_a)][R_b - E(R_b)]\} \quad (10)$$

To analyze the covariance between two stocks it is important to create a DataFrame. The process for calculating the covariance is as follows:

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Establishing dates of the analysis*

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

- *Choosing company and S&P 500*

```
Nutanix = web.DataReader('NTNX', 'yahoo', start, end)
SP500 = web.DataReader('^GSPC', 'yahoo', start, end)
```

- *Calculating returns*

```
Nutanix['Returns'] = Nutanix['Close'].pct_change()
SP500['Returns'] = SP500['Close'].pct_change()
```

- *Remove missing values*

```
Nutanix['Returns'] = Nutanix['Returns'].dropna()
SP500['Returns'] = SP500['Returns'].dropna()
```

- *Create a DataFrame*

```
Nutanix_SP500 = pd.concat([Nutanix['Returns'],SP500['Returns']],axis=1)
Nutanix_SP500.columns = ['Nutanix Returns',' SP500']
```

- *Co-variance Matrix*

```
covariance = Nutanix_SP500.cov()
  covariance
```

	Nutanix Returns	*SP500*
Nutanix Returns	0.001481	0.000119
SP500	0.000119	0.000069

- *Annualized Covariance matrix*

```
annual_covariance = Nutanix_SP500.cov() * 252

annual_covariance
```

	Nutanix Returns	*SP500*
Nutanix Returns	0.373307	0.030005
SP500	0.030005	0.017481

To interpret the covariance, one has to analyze if it is positive or negative, the same way it was done with the variance (Trochim 2020). In this case the daily covariance is positive but the values are small which means that they move together but the relation is not strong. When analyzing the annualized covariance, the same conclusion can be stated. For this it is important to calculate the correlation.

The **correlation** demonstrates how strong the relationship is between two variables, in this case, Nutanix and the S&P500. For determining correlation, the equation is as follows:

Equation 11: Correlation

$$Corr(R_a, R_b) = \frac{Cov(R_a, R_b)}{\sigma(R_a)\sigma(R_b)} \tag{11}$$

Given that the DataFrame *Nutanix_SP500* was created before, the process is simple.

- *Correlation Matrix*

```
correlation = Nutanix_SP500.corr()
  correlation
```

	Nutanix Returns	*SP500*
Nutanix Returns	1.0000	0.3919
SP500	0.3919	1.0000

The correlation coefficient ranges from negative (−1.0) to positive (1.0) (Hayes 2019). When the correlation coefficient is negative (−1.0) it is said that the relation between variables is perfectly negative, which means that they behave contrarily. In the situation that the correlation is positive (1.0) then it is said that they behave perfect positive correlation, which means that they behave in the same way. The rule of thumb of a strong correlation to be considered is if the value is 0.8 or above negative or positive.

In the example of Nutanix, the correlation is positive but it is not strong, since it is below 0.8. Therefore, there is no correlation to be considered between Nutanix and the S&P500.

The process can be elaborated with the *f.fn()* package with certain simplicity. Therefore it is explained as follows:

First, create a process that includes the stocks with the close price.

```
stocks = ffn.get('NTNX:Close, spy:Close', start='2014-01-01', end='2019-
01-01')
stocks.tail()
```

The second step is to create the returns as elaborated in the example before.

```
returns = stocks.to_returns()
returns.tail()
```

	ntnxclose	*spyclose*
Date		
2018-12-24	−0.009323	−0.026423
2018-12-26	0.077221	0.050525
2018-12-27	0.025437	0.007677
2018-12-28	0.009020	−0.001290
2018-12-31	0.032779	0.008759

If the returns that are being planned on using are logarithmic returns, the process is as follows:

```
returns = stocks.to_log_returns()
returns.tail()
```

	ntnxclose	*spyclose*
Date		
2018-12-24	−0.009366	−0.026778
2018-12-26	0.074385	0.049290
2018-12-27	0.025119	0.007648
2018-12-28	0.008980	−0.001291
2018-12-31	0.032253	0.008721

The process will continue with the logarithmic returns. To obtain the correlation matrix is as follows:

	ntnxclose	*spyclose*
ntnxclose	1.000000	0.389009
spyclose	0.389009	1.000000

Scatterplots and Heatmaps

There are two important tools when analyzing the process of comparing stocks. The first one is the scatterplot which will demonstrate the relation between variables, in this case the SP500 and Nutanix (Fig. 14).

To create a scatterplot with a histogram and therefore simplify the process of creating a histogram from scratch *pandas* has a *scatter_matrix* for plotting that can be visualized by using the following command:

- *Creating a scatter matrix*

```
from pandas.plotting import scatter_matrix
scatter_matrix(Nutanix_SP500,figsize=(12,10),alpha=1.0,hist_kwds={'bins':90});
```

The histogram can be adapted by changing the number of bins in the scatter matrix and the alpha can be changed for more transparency.[8] If

[8] For more information on scatter_matrix please visit: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.plotting.scatter_matrix.html.

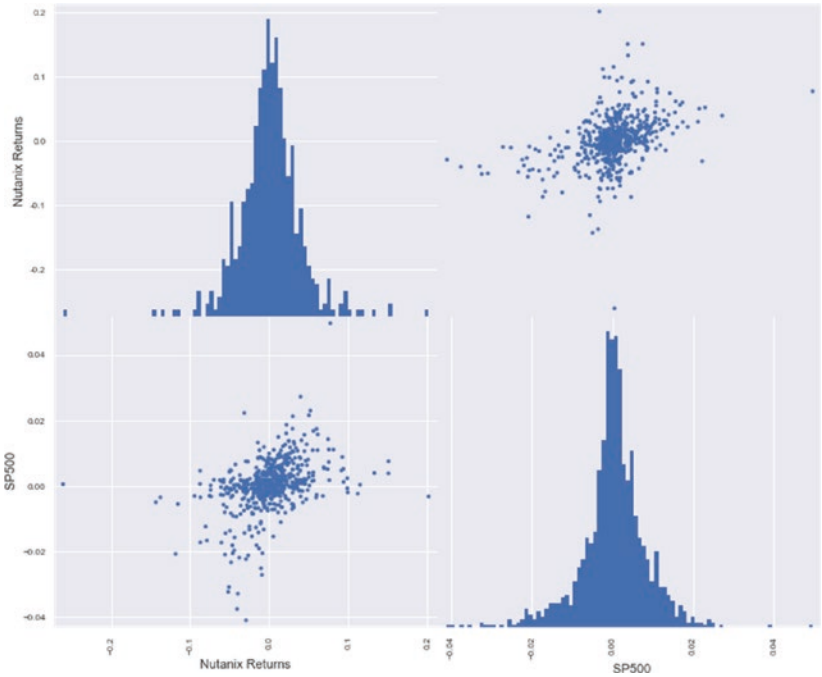


Fig. 14 Scatter Matrix of Nutanix and SP500 (*Source* Elaborated by the author with information from Yahoo Finance)

the only thing needed is the scatterplot, then the process should be as follows (Fig. 15):

- *Creating a scatter plot*

```
Nutanix_SP500.plot(kind='scatter',x='Nutanix Returns',y=' SP500',alpha=0.4,figsize=(15,10));

_ = plt.title('Nutanix and SP500 Scatterplot')
```

Another interesting way of demonstrating the relation between financial data is through a heatmap. A heatmap allows to understand the relation between variables and also scale them considering their relation. The process uses the *seaborn* library and is simple. It can be done as follows (Fig. 16):

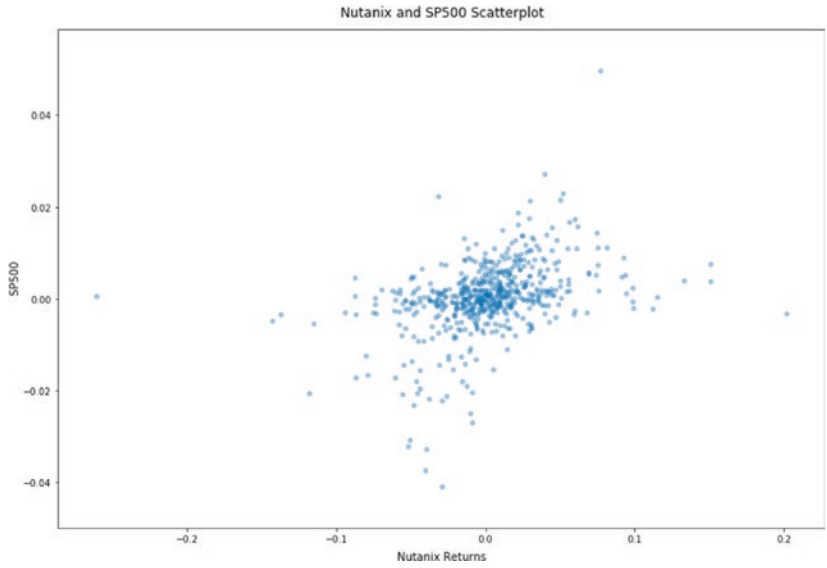


Fig. 15 Nutanix and SP500 Scatterplot (*Source* Elaborated by the author with information from Yahoo Finance)

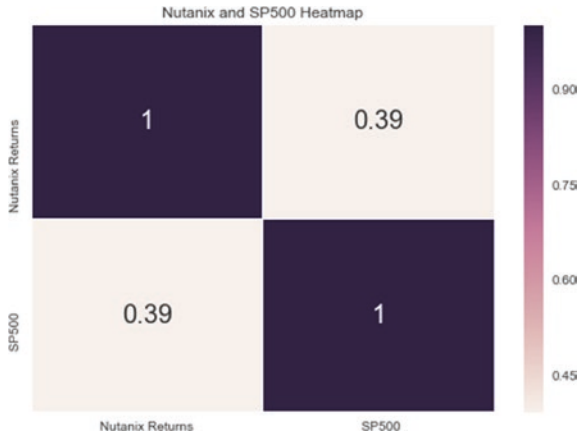


Fig. 16 Nutanix and SP500 Heatmap (*Source* Elaborated by the author with information from Yahoo Finance)

- *Creating a heatmap*

```
import seaborn as sns

sns.heatmap(correlation,annot=True,cmap=None, linewidths=0.3,annot_kws={"size": 20});

_ = plt.title('Nutanix and SP500 Heatmap')
```

The heatmap is a very useful tool since the color demonstrates the comparison between the different variables and the correlation is expressed in the boxes. When using different variables, which will the case of the following chapters, it is important to understand which visual representation is the most appropriate.

The process can be followed in *f.fn()* package in the following process:

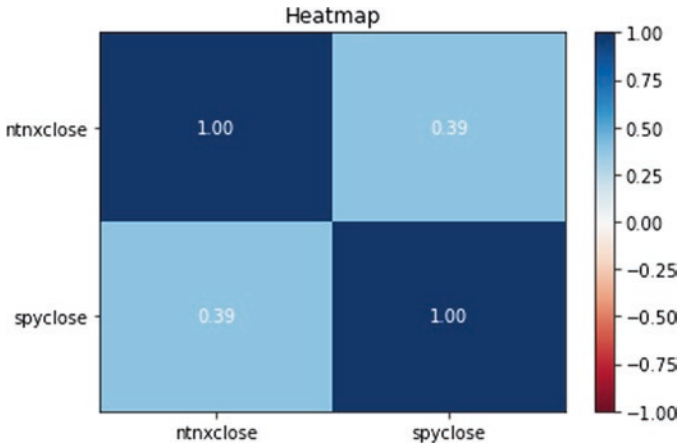
```
returns = stocks.to_log_returns()
returns.tail()
```

	ntnxclose	*spyclose*
Date		
2018-12-24	−0.009366	−0.026778
2018-12-26	0.074385	0.049290
2018-12-27	0.025119	0.007648
2018-12-28	0.008980	−0.001291
2018-12-31	0.032253	0.008721

```
returns.corr()
```

ntnxclose	*spyclose*	
ntnxclose	1.000000	0.389009
spyclose	0.389009	1.000000

```
returns.plot_corr_heatmap();
```



Works Cited

Brooks, Chris. 2008. *Introductory econometrics for finance*. Boston: Cambridge University Press.

Ganti, Akhilesh. 2019. *Central Limit Theorem (CLT)*. 13 September. Accessed April 2, 2019. https://www.investopedia.com/terms/c/central_limit_theorem.asp.

Hargrave, Marshall. 2020. *Standard deviation definition*. 1 February. Accessed February 20, 2020. https://www.investopedia.com/terms/s/standarddeviation.asp.

Hayes, Adam. 2019a. *Correlation definition*. 20 June. Accessed January 1, 2020. https://www.investopedia.com/terms/c/correlation.asp.

Hayes, Adam. 2019b. *Correlation definition*. 20 June. Accessed October 8, 2019. https://www.investopedia.com/terms/c/correlation.asp.

Jain, Diva. 2018. *Skew and Kurtosis: 2 Important statistics terms you need to know in Data Science*. 23 August. Accessed August 12, 2019. https://codeburst.io/2-important-statistics-terms-you-need-to-know-in-data-science-skewness-and-kurtosis-388fef94eeaa.

Kalla, Siddharth. 2020. *Range (Statistics)*. n.d. Accessed January 4, 2020. https://explorable.com/range-in-statistics.

Keaton, Will. 2019. *Quantitative Analysis (QA)*. 18 April. Accessed January 15, 2020. https://www.investopedia.com/terms/q/quantitativeanalysis.asp.

Kenton, Will. 2019. *Kurtosis*. 17 February. Accessed July 30, 2019. https://www.investopedia.com/terms/k/kurtosis.asp.

Trochim, William M.K. 2020. *Correlation*. 10 March. Accessed March 12, 2020. https://conjointly.com/kb/correlation-statistic/.

Wan, Xiang, Wengian Wang, Jiming Liu, and Tiejun Tong. 2014. *Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range*. 19 December. Accessed January 03, 2019. https://bmcmedresmethodol.biomedcentral.com/articles/https://doi.org/10.1186/1471-2288-14-135.

Elements for Technical Analysis Using Python

Abstract Technical Analysis is useful for analyzing the behavior of securities based on different plots with the purpose of determining the upward or downward trends. In the present chapter the discussion will center on returns, volumes, candlestick charts, line charts, simple moving average, MACD, RSI and other technical analysis tools.

Keywords Central limit theorem · Returns · Plots · Statistical measures

In the last chapter, the retrieving of data was explored through the process of using an API or an Excel file. Once the data is on the Jupyter Notebook the next step is to proceed with analyzing the information. The first step is to understand how to display data in Python by using different methods that it provides.

The Linear Plot with One Stock Price (Max & Min Values and the Range)

The first plot analyzed in the book is the **linear plot**. In finance, the linear plot is useful to understand the trend of the data that is being used. The linear plot is an excellent visualization tool, although it only

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_6

demonstrates how the variable has behaved. In the case of using a stock price, the linear plot will show how one of the characteristics of the variable have behaved through a period of time. An example is as follows:

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Setting the continuous data range*

```
start = datetime.datetime(2015,1,1)
end = datetime.datetime(2019,1,1)
```

- *Creating the variable*

```
Tesla = web.DataReader('TSLA','yahoo',start,end)
```

Once the variable is created, the *plt.plot* function is going to be used to create the variable. The *matplotlib.pyplot* also has the possibility of creating a label for the x-axis, a label for the y-axis, and the title for the plot. To do this, the easiest way is to create a *dummy* variable by using the underscore (_). The underscore allows us to add features to a plot without altering the plot. For example (Fig. 1):

```
_ = plt.plot(Tesla['Close'])

_ = plt.xlabel('Date')

_ = plt.ylabel('Closing Price')

_ = plt.title('Tesla Closing Price')
```

There is another way that the linear plot can be elaborated and according to the author it is easier to work with the problem of the plot above that is how the Date is shown. The recommended way to build a linear plot is as follows (Fig. 2):

Fig. 1 Tesla closing price using Python (*Source* Elaborated by the author with information from Yahoo Finance)

Fig. 2 Tesla Closing price with different size (*Source* Elaborated by the author with information from Yahoo Finance)

```
Tesla['Close'].plot(label='Tesla',figsize=(19,8),title='Tesla Closing Price')

_ = plt.xlabel('Date')

_ = plt.ylabel('Closing Price')
```

The above chart uses *figsize* used the width as the first number and the height in inches. The *plt.xlabel* and *plt.ylabel* are used to add the labels to the x and y-axis. These are attributes that are helpful for creating a clean-looking linear plot.

When analyzing the data, the lowest closing price seems to be between January 2016 and July 2017. To find out when the price was the lowest, *argmin* can be used to return the minimum value.

```
Tesla['Close'].argmin()
```

- Result:

```
Timestamp('2016-02-10 00:00:00')
```

As the timestamp demonstrates, the lowest closing price for Tesla was on the 10th of February. To know the price the stock was trading on this day:

```
Tesla['Close'].min()
```

- Result:

```
143.66999799999999
```

The same can be done for the highest closing price traded by Tesla. Given the chart, it is difficult to analyze when was the highest value presented. To establish the date *argmax* can be used on the series:

```
Tesla['Close'].argmax()
```

- Result:

```
Timestamp('2017-09-18 00:00:00')
```

To find out what the highest price in the date range, the function *max* can be used.

```
Tesla['Close'].min()
```

- Result:

```
385.0
```

With the information used by *max* and by *min*, we can obtain the **range.** The range is important. It tells us the difference between the highest and the lowest values. The **range** can be obtained by the difference between *max* and *min* as follows:

```
Tesla['Close'].max() - Tesla['Close'].min()
```

- Result:

```
241.33000200000001
```

When to Use Linear Plots in Finance

Linear plots or line graphs are very useful when analyzing the performance of a stock based on its price. Line plots are one of the most useful resources when representing data such as securities because it allows a comparison between two or more securities (Halton 2019).

The linear plot gives the possibility of visualizing the security over a specific period of time, with a reduction in noise based on that they are elaborated using the closing price (Chen, Line Chart 2019). The use of the closing price is based on the fact that it is the last price that the security has traded at the end of the day in the market.

Using closing prices for security analysis is useful because there is no alteration in the price once it is closed and it allows a more efficient analysis. Considering the highest prices of the day, the lowest prices of the day, the opening price (first price of the day) and the closing price of the day, the last one is the more commonly used by analysts (Kevin 2015).

An example of two or more stocks is extremely useful for highlighting the possibility of analysis.

The Linear Plot with Two or More Stock Price

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Setting the continuous data range*

```
start = datetime.datetime(2019,1,1)
end = datetime.datetime(2020,3,25)
```

- *Creating the variable*

```
American_Airlines = web.DataReader('AAL','yahoo',start,end)
Delta_Airlines = web.DataReader('DAL','yahoo',start,end)
United_Airlines = web.DataReader('UAL','yahoo',start,end)
Southwest_Airlines = web.DataReader('LUV','yahoo',start,end)
JetBlu_Airlines = web.DataReader('JBLU','yahoo',start,end)
```

- *Creating the chart* (Fig. 3)

```
American_Airlines['Close'].plot(label='AAL',figsize=(19,8),title='Closing Price')
Delta_Airlines['Close'].plot(label='DAL')
United_Airlines['Close'].plot(label='UAL')
Southwest_Airlines['Close'].plot(label='LUV')
JetBlu_Airlines['Close'].plot(label='JBLU')

_ = plt.xlabel('Date')

_ = plt.ylabel('Closing Price')

plt.legend();
```

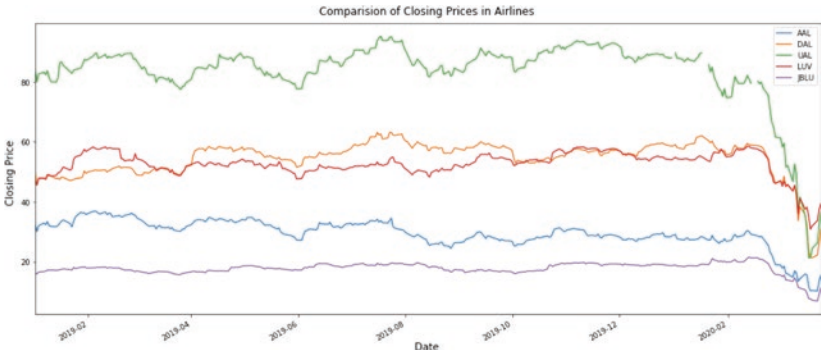


Fig. 3 Comparison of closing prices in Airline Industry (*Source* Elaborated by the author with information from Yahoo Finance)

One of the main differences with the first linear plot is the use of the command *plt.legend()*. The command is useful when there are different variables and the user wants to know which line is related to each security. There are different means of creating a legend, considering the necessity of the user.

- *Examples of different legends*
  - plt.legend((fancybox=True)) adds a shadow around the box.
  - plt.legend((frameon=False,loc='lower center')) locates the legend in the lower center of the graph.
  - plt.legend((frameon=False,loc=upper center')) locates the legend in the upper center of the graph.
  - plt.legend((frameon=False,loc='upper left')) locates the legend in the upper left of the graph.
  - plt.legend((frameon=False,loc='upper right)) locates the legend in the upper right of the graph.
  - plt.legend((frameon=False,loc='upper right, ncol=3)) locates the legend in the upper right and organizes the different companies in three columns. The number of columns can be changed.
  - plt.legend(
    (fancybox=True, framealpha=1, shadow=False, borderpad=1, loc='lower center',ncol=3)
    ,ncol=3);
    fancybox creates a box around the different items of the legend.

Fig. 4 Setting the legend with shadow, framealpha, fancybox and borderpad (*Source* Elaborated by the author with information from Yahoo Finance)

framealpha sets the different shades from white to black. 0 sets the legend in black and 1 in white.

shadow when it is set to `False` eliminated a shadow outside the box and when it is set to `True` it sets a shadow around the box.

borderpad sets the size of the legend box (Fig. 4).

Linear Plot with Volume

The linear plot is not only useful for security prices but it can be also used for identifying trading volume. The trading volume, better known in finance as volume, is the number of contracts or shares that are traded during a period of time (Hayes, Volume Definition 2018). Each transaction in the market is quantified by volume, making it one of the most useful measures of technical analysis. The analysis of the volume has to be done by comparison with the price of the security. If a higher price is also followed by a higher volume, the rising of the price is significant.

It is useful to identify if there is a momentum, which can confirm a trend, or if there is low activity in the market. The example is a continuity of the Tesla line plot. The variable *Tesla* that was created has five categories (1) open, (2) high, (3) low, (4) Close, (5) Adjusted Close and (6) Volume.

- Creating a Volume Plot (Fig. 5)

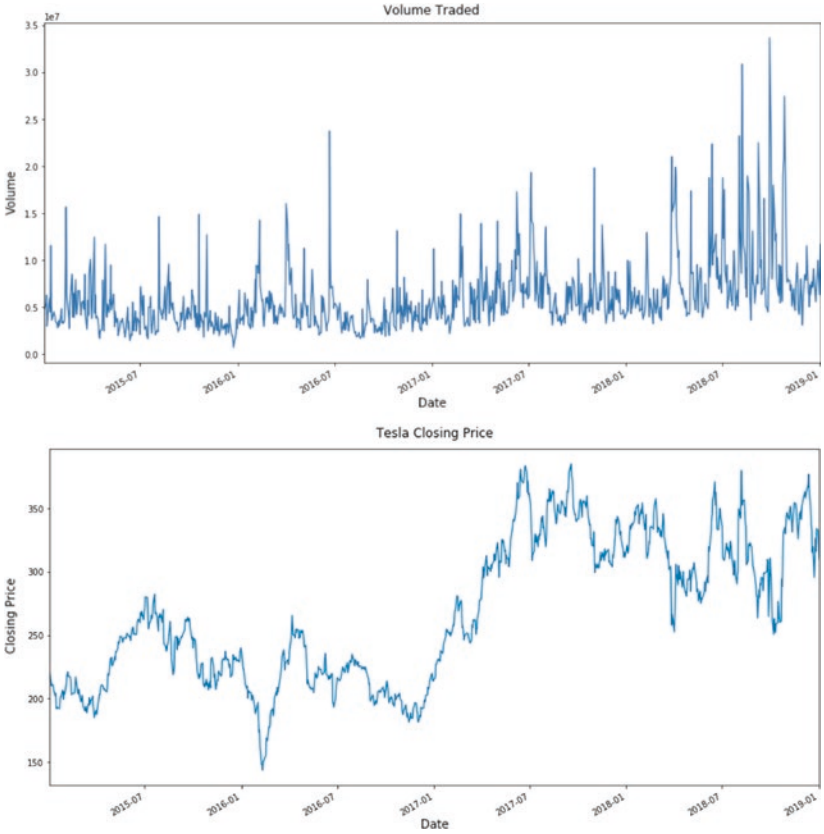


Fig. 5 Comparison of Volume and Closing price of Tesla (*Source* Elaborated by the author with information from Yahoo Finance)

```
Tesla['Volume'].plot(label='Tesla',figsize=(16,8),title='Volume Traded')

_ = plt.xlabel('Date')
_ = plt.ylabel('Volume')
```

When compared, the volume seems to have a movement similar to the closing price of the stock market, assuring that the move is significant.

When the price has been higher there has been more movement in the stocks concerning Tesla.

Volume of Trade

One of the most interesting aspects when analyzing volume is to multiply it by the security price, giving as a result the volume of trade during a specific period in terms of money invested. The total money traded is useful to know the quantity of investment in a single day or period, which is useful when analyzing if the market is selling or buying specific security, or to understand the monetary impact of trade. The equation is rather simple:

Equation 1: Total Money Traded

$$\text{Total Money Trade} = \text{Volume} \times \text{Security Price} \tag{1}$$

- *Obtaining Total Money Traded*

```
Tesla_total_traded = Tesla['Close'] * Tesla['Volume']
```

- *Plotting the Total Money Traded* (Fig. 6)

```
Tesla_total_traded.plot(label='Tesla',figsize=(16,8),title='Total Traded')

_ = plt.xlabel('Date')

_ = plt.ylabel('Total Traded')
```

The *Total Money Traded Plot* demonstrates a similar tendency to the *Volume* plot, but it also identifies was higher. Meaning that the *Total Traded Plot* can be seen as the real growth since it includes the volume of transactions and the current price of the transactions. The *Total Traded Plot* is better used when compared with other companies.

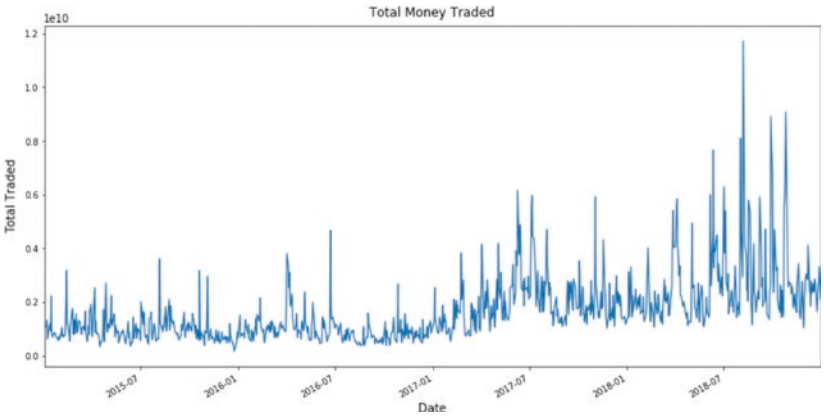


Fig. 6 Total Traded Plot (*Source* Elaborated by the author with information from Yahoo Finance)

Comparison of Securities with Volume Plots and Closing Prices

Comparing volume between securities is vital for financial analysis, basically because it allows the trader to understand trading confirmation, exhaust moves and volume, bullish signs to name a few (Mitchell, How to Use Volume to Improve Your Trading 2020).

To create a plot comparing volumes between securities, the process is as follows:

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Setting the continuous data range*

```
start = datetime.datetime(2019,1,1)
end = datetime.datetime(2020,3,25)
```

- *Choosing securities*

```
Tesla = web.DataReader('TSLA','yahoo',start,end)
General_Motors = web.DataReader('GM','yahoo',start,end)
Ford = web.DataReader('F','yahoo',start,end)
```

- *Plotting the volume* (Fig. 7)

```
Tesla['Volume'].plot(label='Tesla',figsize=(16,8),title='Volume Traded')
General_Motors['Volume'].plot(label='GM')
Ford['Volume'].plot(label='F')

_ = plt.xlabel('Date')

_ = plt.ylabel('Volume')

plt.legend(fancybox=True, framealpha=1, shadow=False, borderpad=1, loc='upper left',ncol=3);
```

As a conclusion of the graph it can be observed that the volume of *Ford* demonstrates that it has been traded considerably more than *Tesla* and *General Motors*. The comparison could be clearer when compared to the price (Fig. 8).

When comparing the closing price where Ford and General Motors basically have behaved as stable, the volume of Ford has been exceedingly variable. Given that Tesla is reasonably higher than Ford, the comparison of the present graph does not explain the reason between volume and price (Fig. 9).

During the last five years, Ford´s security has had a bearish pattern where the investors are believing less and less in the security. This belief that the security will fall has caused a downtrend. The downtrend

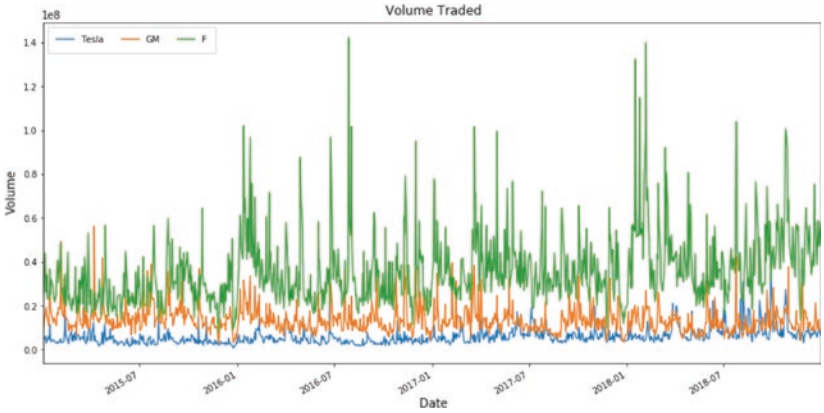


Fig. 7 Volume traded of Tesla, General Motors and Ford (*Source* Elaborated by the author with information from Yahoo Finance)

Fig. 8 Closing Price of Ford, General Motors and Tesla (*Source* Elaborated by the author with information from Yahoo Finance)

explains the volume in which the investors could see an opportunity on selling or buying (Fig. 10).

As seen in Fig. 21, the effect of the volume rises, and the effect is a momentaneous rise in the stock price. This effect is important when analyzing trade because the volume confirms the trends (circle in red) but

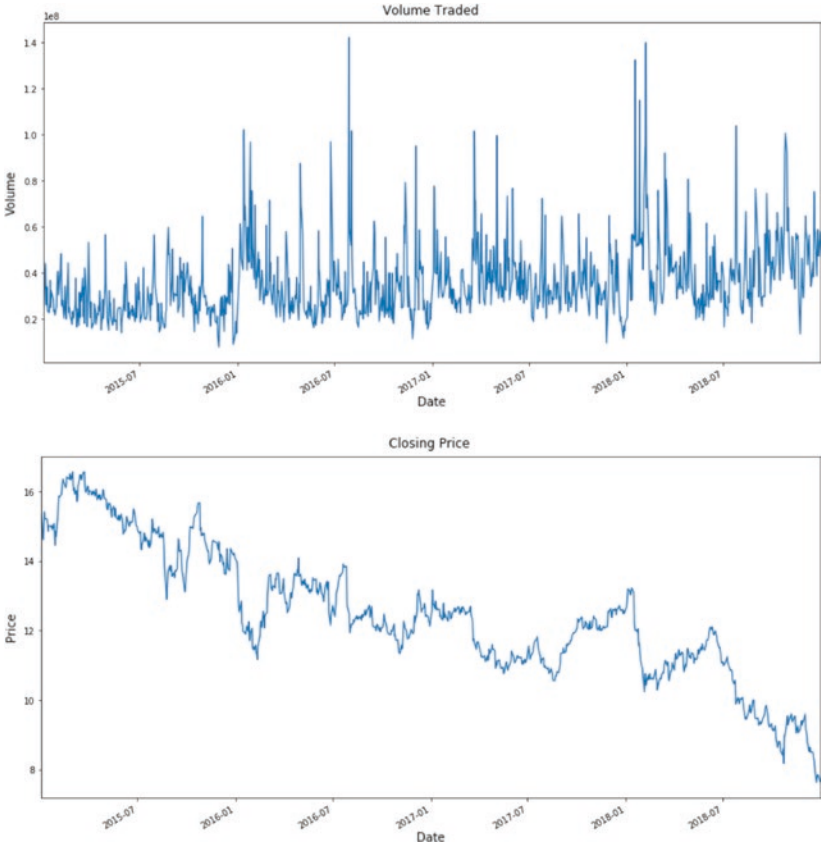


Fig. 9 Comparison of volume and closing price of Ford (*Source* Elaborated by the author with information from Yahoo Finance)

has an important effect on the gains/losses of the investor. The importance of volume is that it can anticipate a price reversal, specifically when with little movement in price there is a strong movement in volume. As a part of technical analysis, it is important to understand this behavior which can be combined with quantitative analysis that will be discussed further in the book.

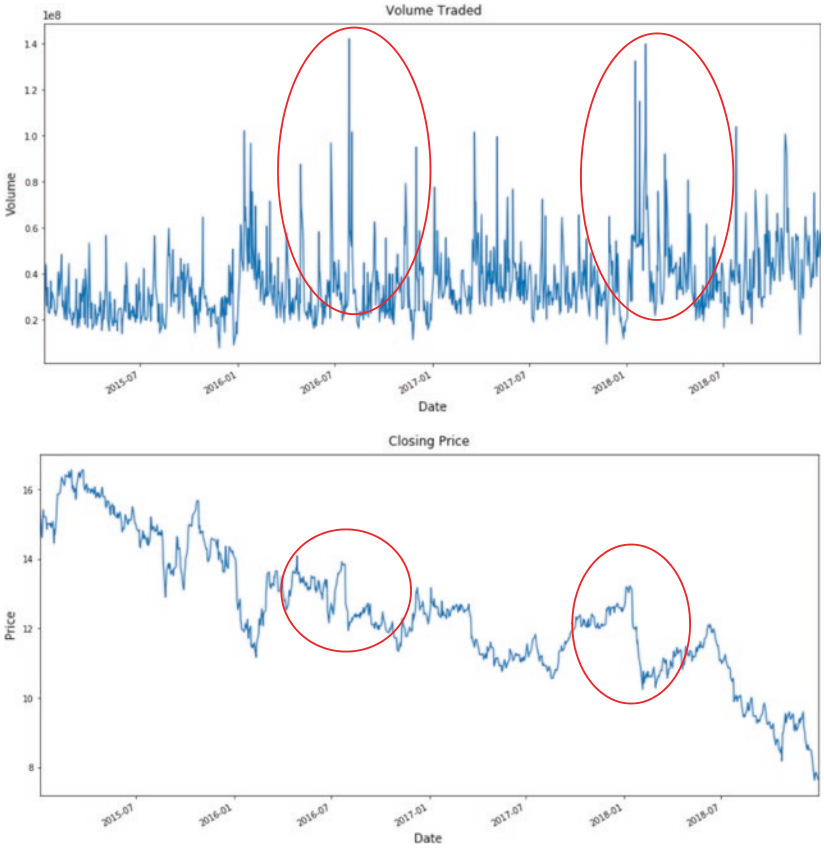


Fig. 10 Understanding volume and price with Ford Security (*Source* Elaborated by the author with information from Yahoo Finance)

Candlestick Charts

The Candlestick charts were created in Japan during the 1700s by Homma whose purpose was to analyze if there was a relation between supply and demand of rice (Mitchell 2019). It is extremely useful for analyzing emotional trading and actually is one of the most useful charts in technical analysis.

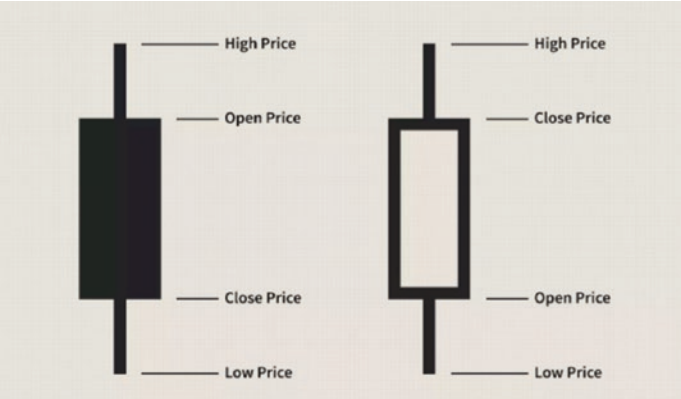


Fig. 11 Candlestick bar representation (*Source* Created by Bang [2019])

The Candlestick chart uses the open, high, low and close price in a day. As stated in Fig. 22, the bar can be filled in or be black, although the colors vary into green and red depending on the user. When the body of the candlestick bar is filled, it means that the close was lower than the open, therefore if the body is empty it means that the close was higher than the open (Fig. 11).

To read a Candlestick chart it is important to understand if it is bullish or bearish. These aspects are based on the price direction by analyzing the close and open prices. It is assumed that the Candlesticks are responsible for the focus on the opening price because of the importance of these charts (J. J. Murphy 1999).

To create a Candlestick chart, the *mplfinance*[1] package will be used. The reason for using this package is the easiness of creating Candlestick charts and how it adapts to the information accessed by Yahoo API.

The *mplfinance* package can be accessed at https://github.com/matplotlib/mplfinance#release. There are different aspects considering this package that is going to be highlighted during the process of creating Candlestick Charts.

[1] For more information concerning the mplfinance package visit: https://github.com/matplotlib/mplfinance/blob/master/examples/customization_and_styles.ipynb.

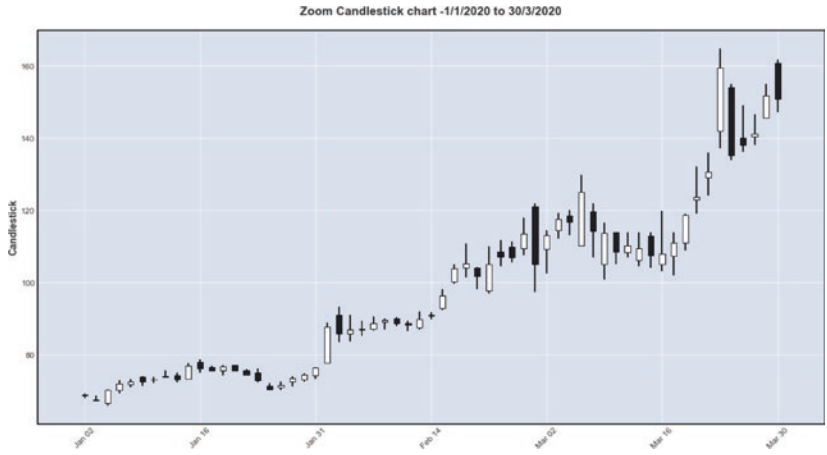


Fig. 12 Zoom candlestick chart (*Source* Elaborated by the author with information from Yahoo Finance)

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import mplfinance as mfp
```

- *Setting the continuousdata range*

```
start = datetime.datetime(2020,1,1)
end = datetime.datetime(2020,3,30)
```

- *Choosing security*

```
zoom = data.DataReader("ZM", 'yahoo', start, end)
```

- *Plotting the candlestick* (Fig. 12)

```
mfp.plot(zoom,type='candle', title ='Zoom Candlestick chart -1/1/2020 to 30/3/2020',
ylabel='Candlestick',figratio=(30,15),figscale=0.75)
```

The above chart analyzes the trend concerning the security Zoom (ticker ZM) which has seen an important growth given the COVID-19 and the lockdown in various countries. From March 16 until March 30 the candles have grown wider because of its high and low price, which demonstrates high volatility in the security. A bullish pattern can also be seen as well as the difference between closing price and opening price, which leads to black and white candlesticks.

- Customizing the plot
  - type='candle' determines the different choices that can be made to create a chart. In the above example, 'candle' was used but there are other options such as 'line', 'ohlc', 'bars' ,'ohcl_bars'.
  - title gives the possibility of creating a title in the graph to describe what the candlestick or other graph is charting. It is important to highlight that there are no year dates in the graphs, therefore creating a title with the expressed years are important.
  - ylabel='Candlestick' sets a label in the y axis.
  - figratio=(30,15) is for the author the ideal size for analysis, the numbers can be altered to create a wider or narrower graph as well as bigger or smaller.
  - figscale=0.75 is a conventional number for using in graphs.

Candlestick Charts and Volume

Candlesticks may also benefit from the analysis of volume. The purpose of analyzing candlesticks and volume is based on the fact that the prices are guided by the transactions, as exposed previously.

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import mplfinance as mfp
```

- *Setting the continuous data range*

```
start = datetime.datetime(2020,1,1)
end = datetime.datetime(2020,3,30)
```

Fig. 13 Dow Jones Candlestick chart with volume (*Source* Elaborated by the author with information from Yahoo Finance)

- *Choosing security*

```
Dow_Jones = data.DataReader("^DJIA", 'yahoo', start, end)
```

- *Plotting the candlestick with volume* (Fig. 13)

```
mfp.plot(Dow_Jones,type='candle', title =Dow Jones Candlestick chart with volume -
1/1/2020 to 30/3/2020', ylabel='Candlestick',figratio=(30,15),figscale=0.75, volume=True) )
```

As Fig. 24 demonstrates, there is a bearish pattern in the stock and when analyzed with the volume, the shares traded have been growing since February 14th, 2020, which is when the COVID-19 began to have an impact in Spain and Italy. Also, on March 12 the World Health Organization declared COVID-19 as a pandemic, which surged the volume and lowered considerably the price. Let us remember that according to the Dow Jones Theory, the market discounts everything and this is expressed in the price.

Customizing Candlestick Charts and Volume with **Kwargs

**Kwargs are extremely useful when working with functions and charts. In the case of the *mplfinance* the kwargs can be useful to add the customization of the plots into an only variable. Use kwargs in charts when there are many variables rather than using an approach of describing each variable, this is helpful for those reading the notebook (Mastromatteo 2020).

- *Creating a **kwarg*

For creating a **kwarg the process is simple because it is based on a dictionary. To create a *kwarg* the process is as follows:

```
kwargs =  dict(type='candle', title ='Zoom Candlestick chart -1/1/2020 to 30/3/2020',
ylabel='Candlestick',figratio=(30,15),figscale=0.75, volume=True) )
```

- *Using the kwarg in a process* (Fig. 14)

```
mfp.plot(Dow_Jones,**kwargs,style='yahoo')
```

The result of using *kwargs* is that it creates a dictionary, giving easy access to the user when adapting the chart with the different formats such as 'starandstripes', 'brasil' , 'mike' , 'charles' and 'classic'. The adaptation is user-friendly and it allows for any alteration of the charts.

OHLC Charts with Volume

The OHCL charts follow the same idea behind the Candlestick charts. Its name is derived from the Open price, High price, Closing price and Lower price. An OHLC chart might be easy for some users to understand when analyzing data, therefore it is included in the present book. To interpret the OHLC chart, one must understand what each bar means (Fig. 15).

When elaborating an OHLC chart using *mplfinance* it is a similar process to the creation of the candlestick charts. Volume can also be added as part of the chart and the customization of the chart is the same as the candlestick.

Fig. 14 Candlestick chart and volume chart using colors (*Source* Elaborated by the author with information from Yahoo Finance)

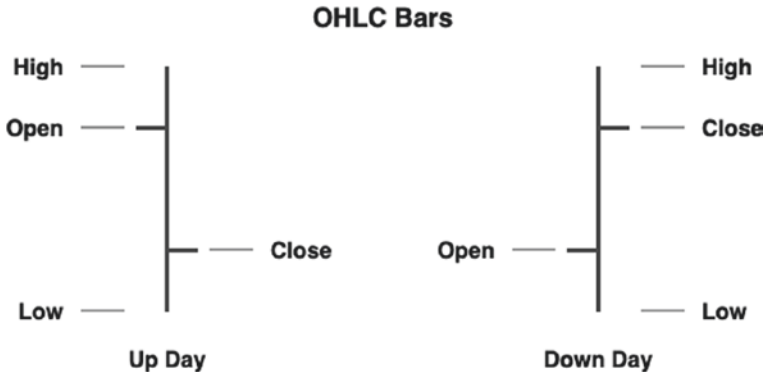


Fig. 15 OHLC bars explained (*Source* From the article written by Basurto [2020])

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import mplfinance as mfp
```

- *Setting the continuous data range*

```
start = datetime.datetime(2020,1,1)
end = datetime.datetime(2020,3,30)
```

- *Choosing security*

```
Dow_Jones = data.DataReader("^DJIA", 'yahoo', start, end)
```

- *Plotting the candlestick with volume* (Fig. 16)

```
kwargs =  dict(title ='Dow Jones OHLC chart -1/1/2020 to 30/3/2020', ylabel='OHLC
chart',figratio=(30,15),figscale=0.75, volume=True) )

mfp.plot(Dow_Jones,**kwargs)
```

As show in Fig. 27 the process and the interpretation are similar to a Candlestick chart. The volume has led to a change in prices and has had an effect on the volatility of the security. The relation between shares traded and the movement in price demonstrates the stated argument.

Line Charts with Volume

In one of the sections discussed earlier using *matplotlib* the analysis of volume and a line chart was created by elaborating two charts. Using *mplfinace* this can be done on the same chart.

- *Importing libraries*

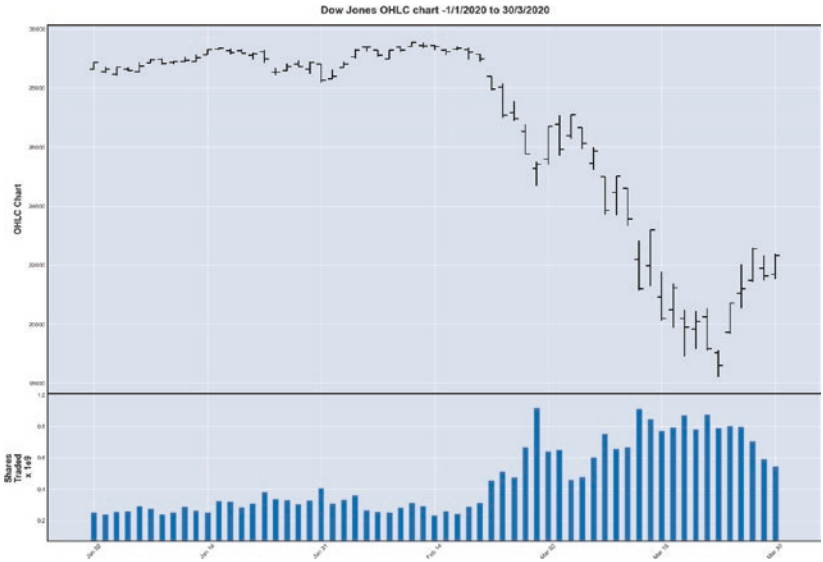


Fig. 16 Dow Jones OHLC chart with volume (*Source* Elaborated by the author with information from Yahoo Finance)

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import mplfinance as mfp
```

- *Setting the continuous data range*

```
start = datetime.datetime(2018,1,1)
end = datetime.datetime(2020,3,30)
```

- *Choosing security*

```
Dow_Jones = data.DataReader("^DJIA", 'yahoo', start, end)
```

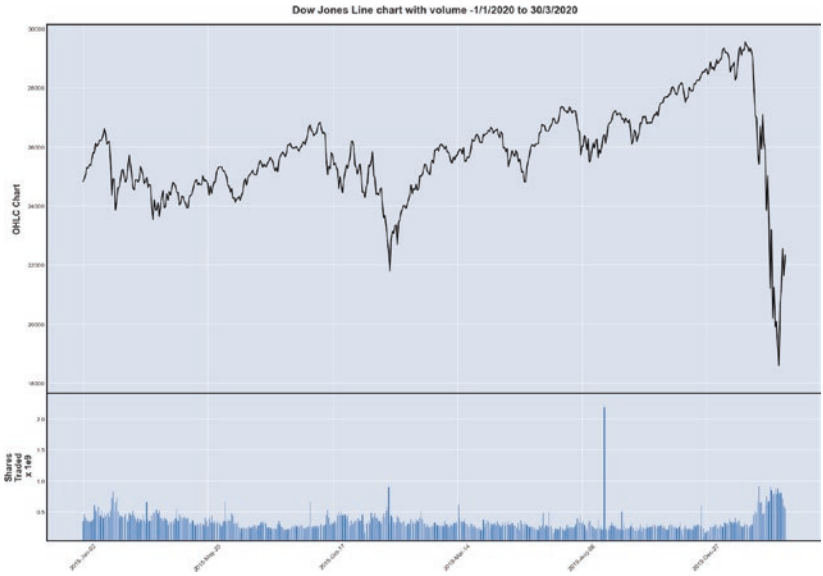


Fig. 17 Line charts with volume (*Source* Elaborated by the author with information from Yahoo Finance)

- *Plotting the candlestick with volume* (Fig. 17)

```
kwargs =  dict(title ='Dow Jones line chart with volume-1/1/2020 to 30/3/2020', ylabel='Line chart',figratio=(30,15),figscale=0.75, volume=True) )

mfp.plot(Dow_Jones,**kwargs, type= 'line')
```

Moving Average with Matplotlib

The moving average is a technical indicator part of the technical analysis, which means that it is centered on the trends of the stocks based on trading activity. It filters the noise concerning short-term prices and is useful to identify a trend direction (Hayes 2020). The moving average (MA) can be divided into simple moving average (SMA) or exponential moving average (EMA). Both of the indicators are useful for elaborating the

Moving Average Convergence Divergence (MACD) which is functional for determining the momentum of a stock.

For creating an SMA, the companies that will be used are Amazon, Walmart and Target. To calculate an SMA the first step is to determine how many periods will be used as the average. The most common periods are 20, 50, 100 and 200 days (Milton 2020). This depends on the data that is available and the purpose of the analysis. If there has been a strong movement in the security market, it may be useful to use the 50-day period. If there has not been any change in the companies during of comparison it may be useful to use the 100- or 200-day period.

- Installing packages

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- Establishing dates of the analysis

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

- Choosing companies

```
Amazon = web.DataReader('AMZN', 'yahoo', start, end)
Walmart = web.DataReader('WMT', 'yahoo', start, end)
Target = web.DataReader('TGT', 'yahoo', start, end)
```

- Establishing the SMA for 50 days

```
Amazon['MA50'] = Amazon['Close'].rolling(50).mean()
Walmart['MA50'] = Walmart['Close'].rolling(50).mean()
Target['MA50'] = Target['Close'].rolling(50).mean()
```

```
Amazon['MA100'] = Amazon['Close'].rolling(100).mean()
Walmart['MA100'] = Walmart['Close'].rolling(100).mean()
Target['MA100'] = Target['Close'].rolling(100).mean()


Amazon['MA200'] = Amazon['Close'].rolling(200).mean()
Walmart['MA200'] = Walmart['Close'].rolling(200).mean()
Target['MA200'] = Target['Close'].rolling(200).mean()
```

For the above example the *rolling*[2] DataFrame from pandas was used. By using *rolling* the window of days can be defined (in this case 50) and by using the rolling with the mean, the result is the average of the 1st day to the 50th day and then the average of 2nd day to the 51st day and so on.

Once the three companies have in the DataFrame the MA50 column, then they can be plotted individually to analyze the momentum of the stocks.

- *Walmart Plot* (Fig. 18)

```
Walmart['MA50'].plot(label='Walmart SMA50',figsize=(16,8))
Walmart['MA100'].plot(label='Walmart SMA100)
Walmart['MA200'].plot(label='Walmart SMA200)
Walmart['Close'].plot(label='Walmart Close')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Walmart SMA50 & Close Price Comparison')

plt.legend();
```

[2] For more information concerning rolling visit: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rolling.html.

Fig. 18 Simple moving average of Walmart (*Source* Elaborated by the author with information from Yahoo Finance)

- *Target Plot* (Fig. 19)

```
Target['MA50'].plot(label='Target SMA50',figsize=(16,8))
Target['MA100'].plot(label='Target SMA100)
Target['MA200'].plot(label='Target SMA200)
Target['Close'].plot(label='Target Close')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Target SMA50 & Close Price Comparison')

plt.legend();
```

Fig. 19 Simple moving average of Target (*Source* Elaborated by the author with information from Yahoo Finance)

- *Amazon Plot* (Fig. 20)

```
Amazon['MA50'].plot(label='Amazon SMA50',figsize=(16,8))
Amazon['MA100'].plot(label='Amazon SMA100)
Amazon['MA200'].plot(label='Amazon SMA200)
Amazon['Close'].plot(label='Amazon Close')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Amazon SMA50 & Close Price Comparison')
```

As seen on the plots, the momentum of Target and Amazon in the selected dates, the SMA50 shows a bigger fall than the SMA 200. It is important to notice that the three SMA predict a fall during the selected dates. This could be seen as an indication that the security could fall. In the case of Walmart, it is different since SMA50 and the SMA200 is that the SMA200 demonstrates a stability in the stock.

Fig. 20 Simple moving average of Amazon (*Source* Elaborated by the author with information from Yahoo Finance)

Moving Average with Mplfinance

Using the *mplfinance* for creating SMA is far more user-friendly than *matplotlib*. The only inconvenience with the SMA in *mplfinance* is that at the time this book is published, there are no options for adding legends.[3]

Other than the mentioned aspect, the process adds the characteristic *mav* for moving average in which the number represents the different periods.

- *Importing libraries*

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import mplfinance as mfp
```

[3] The author contacted Daniel Goldfarb who is in charge of *mplfinance* and addressed the issue in the following link: https://github.com/matplotlib/mplfinance/issues/21s.

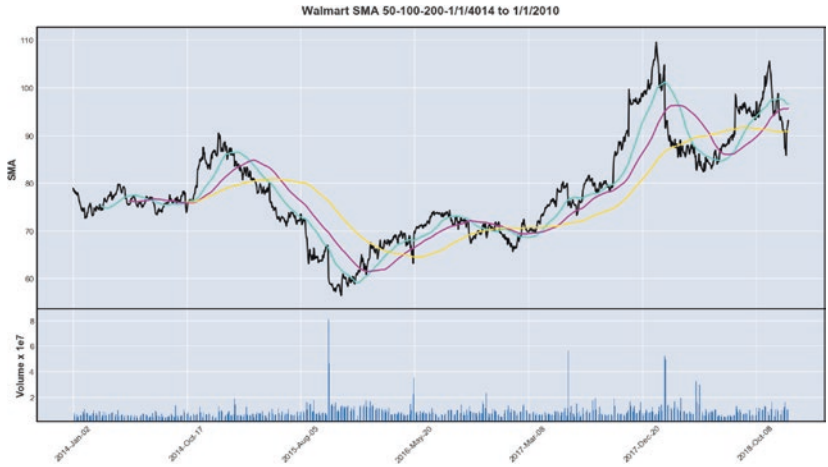


Fig. 21 Simple moving average for Walmart (*Source* Elaborated by the author with information from Yahoo Finance)

- *Setting the continuous data range*

```
start = datetime.datetime(2014,1,1)
end = datetime.datetime(2019,1,1)
```

- *Choosing security*

```
Walmart = web.DataReader("^DJIA", 'yahoo', start, end)
```

- *Plotting the candlestick with volume* (Fig. 21)

```
kwargs =  dict(title = 'Walmart SMA 50-100-200-1/1/4014 to 1/1/2010',
ylabel='SMA',figratio=(30,15),figscale=0.75, volume=True) )

mfp.plot(Walmart,**kwargs, type= 'line')
```

The SMA is equivalent to the one elaborated with *matplotlib* in Fig. 19: Simple moving average of Walmart. The only problem are the legends,

but the process is far simpler. For a quick analysis it is recommended to use the *mplfinance* as the main resource.

The Exponential Moving Average (EMA)

The exponential moving average (EMA) has the same background as the moving average but it gives a specific weight to recent data points, which is the most important difference. The EMA is also a technical indicator that is useful to identify trends in the security market. The 50, 100 and 200 days are also used in the EMA as conventionalism (Hayes 2020).

When programming the EMA the Pandas DataFrame *ewm* for exponential weighted is combined with the *mean*. This is an important aspect since the EMA has an exponential weight that will determine the effect on the curves and how it relates to the market. The process is done as follows:

- Installing packages

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- Establishing dates of the analysis

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

- Choosing companies

```
Amazon = web.DataReader('AMZN', 'yahoo', start, end)
Walmart = web.DataReader('WMT', 'yahoo', start, end)
Target = web.DataReader('TGT', 'yahoo', start, end)
```

- Elaborating the EMA

```
Amazon['EMA50'] = Amazon['Close'].ewm(span=50, adjust=False).mean()
Amazon['EMA100'] = Amazon['Close'].ewm(span=100, adjust=False).mean()
Amazon['EMA200'] = Amazon['Close'].ewm(span=200, adjust=False).mean()


Walmart['EMA50'] = Walmart['Close'].ewm(span=50, adjust=False).mean()
Walmart['EMA100'] = Walmart['Close'].ewm(span=100, adjust=False).mean()
Walmart['EMA200'] = Walmart['Close'].ewm(span=200, adjust=False).mean()


Target['EMA50'] = Target['Close'].ewm(span=50, adjust=False).mean()
Target['EMA100'] = Target['Close'].ewm(span=100, adjust=False).mean()
Target['EMA200'] = Target['Close'].ewm(span=200, adjust=False).mean()
```

- Plotting the EMA of the three companies (Figs. 22, 23, and 24)

```
Amazon['EMA50'].plot(label='Amazon EMA50',figsize=(16,8))
Amazon['EMA100'].plot(label='Amazon EMA100')
Amazon['EMA200'].plot(label='Amazon EMA200')
Amazon['Close'].plot(label='Amazon Close')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Amazon EMA & Close Price Comparison')

plt.legend();
```

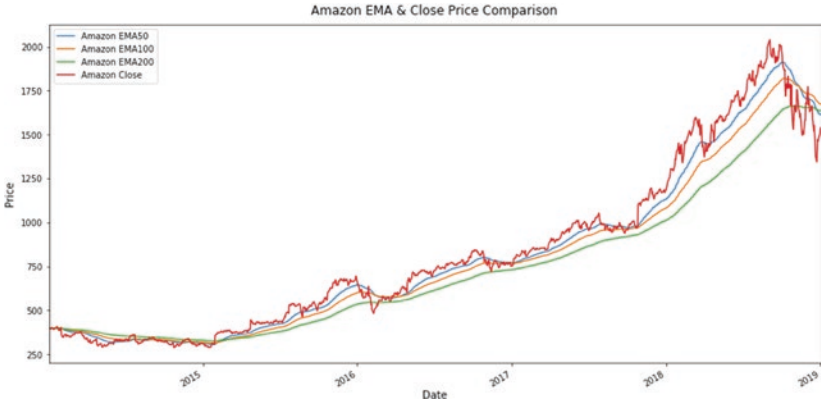


Fig. 22 Amazon EMA 50, 100, 200 (*Source* Elaborated by the author with information from Yahoo Finance)

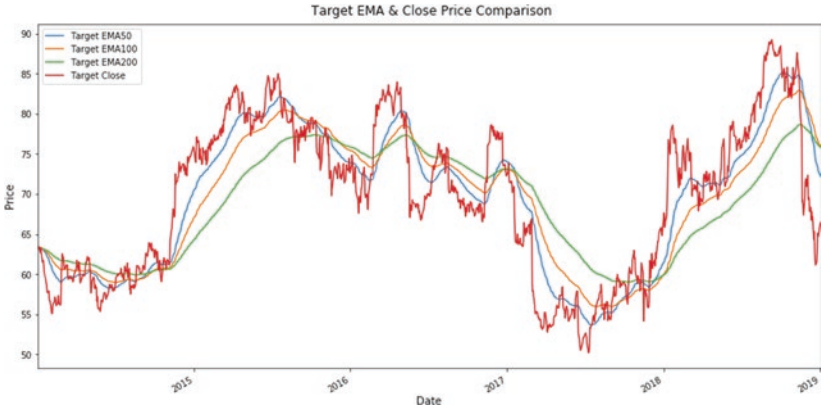


Fig. 23 Target EMA 50, 100,200 (*Source* Elaborated by the author with information from Yahoo Finance)

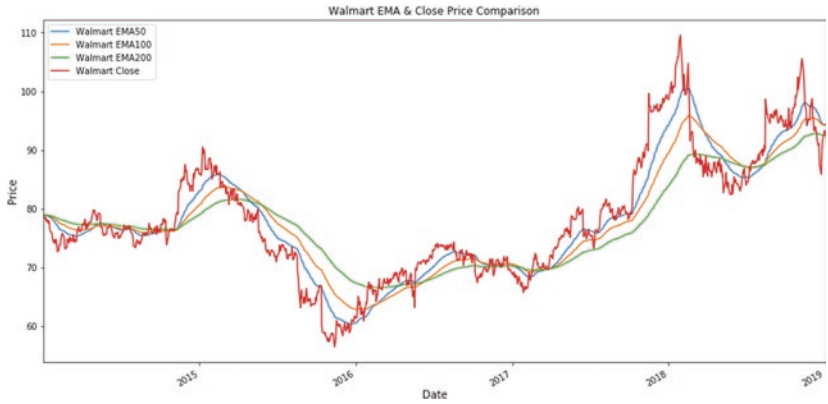


Fig. 24 Walmart EMA 50, 100,200 (*Source* Elaborated by the author with information from Yahoo Finance)

```
Target['EMA50'].plot(label= Target EMA50',figsize=(16,8))
Target ['EMA100'].plot(label= Target EMA100')
Target ['EMA200'].plot(label= Target EMA200')
Target ['Close'].plot(label= Target Close')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title(Target EMA & Close Price Comparison')

plt.legend();
```

```
Walmart['EMA50'].plot(label= Walmart EMA50',figsize=(16,8))
Walmart ['EMA100'].plot(label= Walmart EMA100')
Walmart ['EMA200'].plot(label= Walmart EMA200')
Walmart ['Close'].plot(label= Walmart Close')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title(Walmart EMA & Close Price Comparison')

 plt.legend();
```

When compared with the SMA the difference is that the fall of Target and Amazon is not as drastic as in the SMA. This leads us to choose between selecting the EMA strategy by analyzing the current effects on the market or the SMA if the market has seen an important change in the past few weeks and one thinks it should not have a weight in the analysis.

The Moving Average Convergence Divergence (MACD) with Baseline

The Moving Average Convergence Divergence, better known as MACD, was created by Gerard Appel with the purpose of understanding the market behavior. The MACD is part of technical analysis and one of the most used indicators when trading (Mitchell 2019).

The MACD is composed of three components[4]:

- The MACD line which measures the distance between two moving averages.
- Signal line that identifies price change
- Histogram that represents the difference between MACD and signal line.

[4] For more information visit the article: https://medium.com/duedex/what-is-macd-4a43050e2ca8

For calculating the MACD it is important to first calculate the Exponential Moving Average (EMA). For this, it is useful to use the Pandas DataFrame *ewm* that was used before. The equation to calculate the MACD is the following:

Equation 2: MACD equation with EMA

$$\text{MACD} = \text{EMA for 12 periods} - \text{EMA for 26 periods} \quad (2)$$

To input the equation in Python the *ewm* will beç combined with the *mean* function to obtain the MACD. The span will be set in the first part of the equation in 12 and the second part of the equation in 26. The elaboration in Python is simple and is done as follows:

- Installing packages

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- Establishing dates of the analysis

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

- Choosing companies

```
Amazon = web.DataReader('AMZN', 'yahoo', start, end)
Walmart = web.DataReader('WMT', 'yahoo', start, end)
Target = web.DataReader('TGT', 'yahoo', start, end)
```

- Creating the MACD

```
Amazon['MACD']=Amazon['Close'].ewm(span=12,adjust=False).mean()-
Amazon['Close'].ewm(span=26, adjust=False).mean()
```

```
Walmart['MACD']=Walmart['Close'].ewm(span=12,adjust=False).mean()-
Walmart['Close'].ewm(span=26, adjust=False).mean()

Target['MACD']=Target['Close'].ewm(span=12,adjust=False).mean()-
Target['Close'].ewm(span=26, adjust=False).mean()
```

- Creating the baseline

```
Amazon['baseline'] = 0
Walmart['baseline'] = 0
Target['baseline'] = 0
```

- Plotting the MACD (Figs. 25, 26, and 27)

```
Amazon['MACD'].plot(label='Amazon MACD',figsize=(16,8))
Amazon['baseline'].plot(label='Baseline')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Amazon MACD')

plt.legend();
```

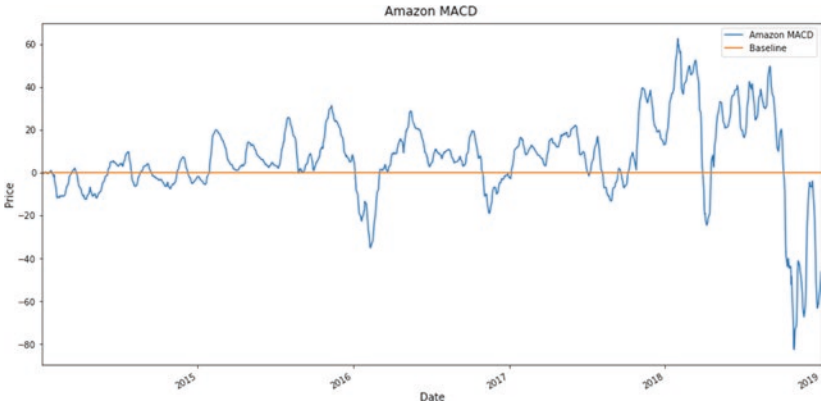


Fig. 25 Amazon MACD with Baseline (*Source* Elaborated by the author with information from Yahoo Finance)

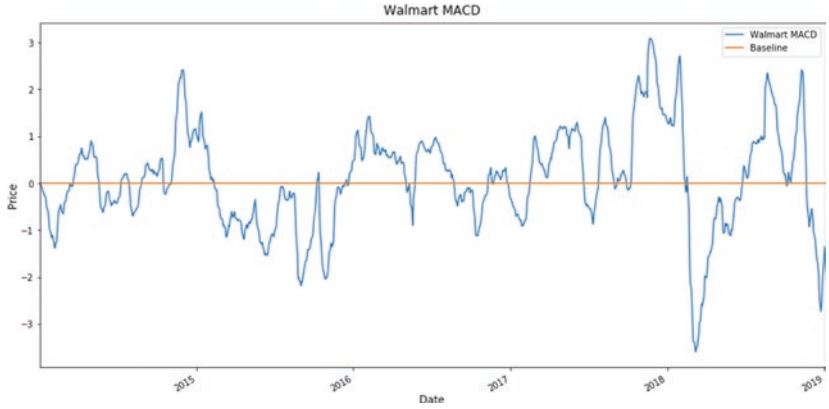


Fig. 26 Walmart MACD with Baseline (*Source* Elaborated by the author with information from Yahoo Finance)

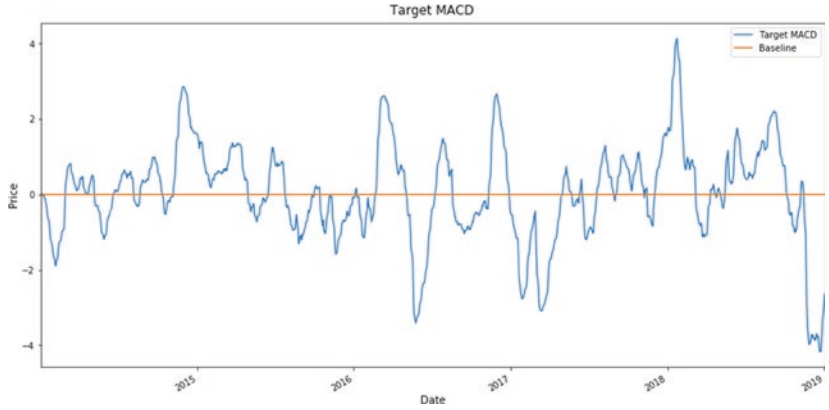


Fig. 27 Target MACD with baseline (*Source* Elaborated by the author with information from Yahoo Finance)

```
Walmart['MACD'].plot(label='Walmart MACD',figsize=(16,8))
Walmart['baseline'].plot(label='Baseline')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Walmart MACD')

plt.legend();


Target['MACD'].plot(label='Target MACD',figsize=(16,8))
Target['baseline'].plot(label='Baseline')

_ = plt.xlabel('Date')

_ = plt.ylabel('Price')

_ = plt.title('Target MACD')

plt.legend();
```

The interpretation of the MACD is set on the baseline which is zero. When the MACD is above the baseline then the stock or market is bullish, if the market is below the zero line then the market is bearish. The MACD is also presented in a histogram but the line graph is a simpler way to illustrate the process.

The Moving Average Convergence Divergence (MACD) with Signal Line

Using a signal line in the MACD is one of the most important trading items because of its interpretation.

- When the MACD crosses the signal line from below to above the indicator is considered bullish.
- When the MACD crossed the signal line from above to below the indicator is considered bearish (Posey 2019).

The signal line is equivalent to an EMA of nine periods. The signal line will be plotted with the MACD.

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Establishing dates of the analysis*

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)
```

- *Choosing companies*

```
Target = web.DataReader('TGT', 'yahoo', start, end)
```

- *Creating the EMA for 12 and 26 periods*

```
ema_12 = Target['Close'].ewm(span=12, adjust=False).mean()
ema_16 = Target['Close'].ewm(span=26, adjust=False).mean()
```

- *MACD and Signal Line (9 periods)*

```
macd = ema_12 - ema_16
signal_line = macd.ewm(span=9, adjust=False).mean()
```

- *Plotting the MACD and the Signal Line* (Fig. 28)

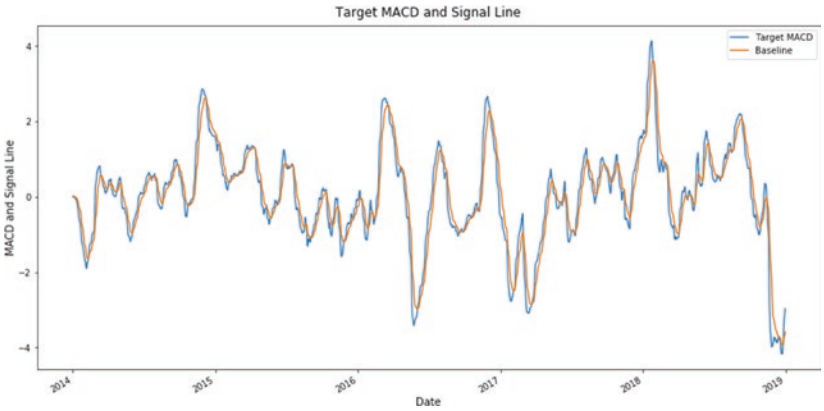


Fig. 28 MACD and Signal Line (*Source* Elaborated by the author with information from Yahoo Finance)

```
macd.plot(label='Target MACD',figsize=(16,8))
signal_line.plot(label='Baseline')

_ = plt.xlabel('Date')

_ = plt.ylabel('MACD and Signal Line')

_ = plt.title('Target MACD and Signal Line')

plt.legend();
```

Bollinger Bands ®

When John Bollinger created the Bollinger Bands in the 1980s he found a means of joining the quantitative aspect (standard deviation) and technical analysis for decision-making. Bollinger bands combine standard deviation, a measure for volatility, and moving average defining when the security has a contraction or an expansion (Bollinger 2018).

Bollinger Bands are extremely useful when analyzing security because:

- When there is low volatility the bands will be close together. When there is high volatility the bands will be apart. Periods of low volatility are often followed by high volatility. The same is for periods of high volatility followed by low volatility.
- If a price moves beyond the upper barrier the prices are considered overbought. Meaning that the stock is being bought at unjustifiably high prices.
- If a price moves below the upper barrier then the prices are considered oversold. Meaning that the stocks are selling below its true value.

To calculate a Bollinger Band there is a need for a window of days for the average and a number of standard deviations for the higher and lower bands. The number of days conventionally is set to 20 days. For the Bollinger Band the package that will be used is the *mplfinance*.

- Importing libraries

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import datetime
import mplfinance as mfp
```

- Setting the continuous data range

```
start = datetime.datetime(2018,8,1)
end = datetime.datetime(2020,3,30)
```

- Choosing security

```
Dow_Jones = data.DataReader("^DJIA", 'yahoo', start, end)
```

- Creating the Bollinger Bands

```
window_of_days = 20
number_std = 2


rolling_mean = Dow_Jones['Close'].rolling(window_of_days).mean()
rolling_std = Dow_Jones['Close'].rolling(window_of_days).std()


Dow_Jones['Rolling Mean'] = rolling_mean
Dow_Jones['Bollinger High'] = rolling_mean + (rolling_std * number_std)
Dow_Jones['Bollinger Low'] = rolling_mean - (rolling_std * number_std)
```

- Plotting the Bollinger Bands (Fig. 29)

```
high_low = Dow_Jones[['Bollinger Low','Bollinger High']]

apd  = mfp.make_addplot(high_low)

kwargs = dict(title ='Dow Jones Bollinger Band 1/8/2019 to 1/4/2020', ylabel='Bollinger Bands',
figratio=(30,15),figscale= 0.75)

mfp.plot(Dow_Jones, addplot = apd, **kwargs, type = 'line')
```

As expressed before, there are three breaking points in the lower band during the COVID-19 pandemic crisis, which means that the stocks are oversold. This leads to the argument that because of fear there was *dumping* or selling securities before they lost value.

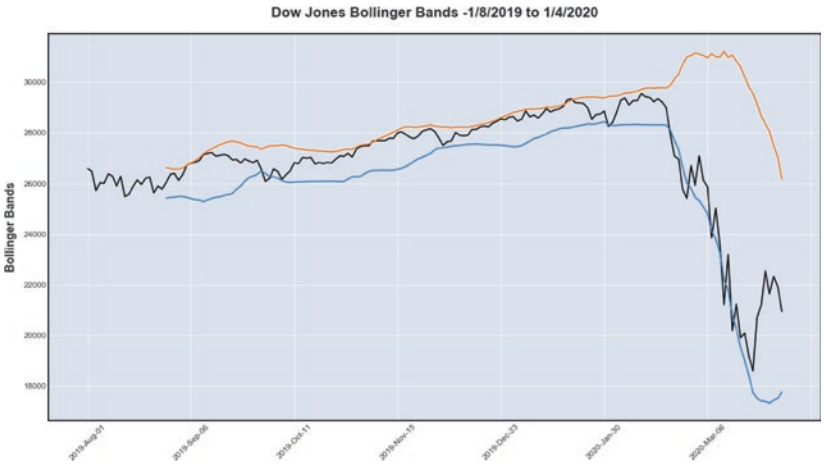


Fig. 29 Dow Jones Bollinger Bands (*Source* Elaborated by the author with information from Yahoo Finance)

Backtesting Strategies for Trading

Backtesting has become extremely useful with Python and API data because it allows historical information to be accessed easily, as seen before, and be used nn a prompt manner. Backtesting for trading is important because it offers a strategy for creating returns based on the performance of a stock.

Parabolic SAR

The parabolic SAR gives and edges to the traders given that it analyzes the movement of the stock. It was created by J. Welles Wilder Jr., which also created the RSI (C. Murphy 2020). The logic behind the parabolic SAR is as follows:

Equation 3: Uptrend and Downtrend SAR Equation

$$\begin{aligned}&\text{Uptrend Parabolic SAR} = \text{Prior SAR}\\ &+ \text{Prior Acceleration Factor} * (\text{Prior Extreme Point} - \text{Prior SAR})\end{aligned}$$

$$\text{Downtrend Parabolic SAR} = \text{Prior SAR} - \text{Prior Acceleration Factor} * (\text{Prior SAR} - \text{Prior Extreme Point}) \tag{3}$$

There are important aspects when creating a parabolic SAR, for example, the extreme point in the uptrend is the highest price, usually refer as *High*, and in the Downtrend, it is the low price referred to as *Low*. The acceleration factor is usually set at 0.02. The acceleration factor affects the SAR when the extreme point is recorded by 0.02. The acceleration factor is modified depending on each trader and its strategy.

When using the TA-Lib library for calculating the parabolic SAR the function *SAR* will be used to retrieve the information. It will ask for the following:

- Choose the *High* price for the extreme points in uptrend
- Choose the *Low* price for the extreme point in downtrend.
- Choose the acceleration factor (as recommended 0.02)
- Choose the maximum for the acceleration factor (recommended 0.02)

The implementation is as follows:

- Import packages including talib

```
import talib
import numpy as np
import pandas as pd
import pandas_datareader as pdr
from pandas_datareader import data as web
import datetime
import mplfinance as mpf
import matplotlib.pyplot as plt
%matplotlib inline
```

- Choose dates

```
start = datetime.datetime(2019,8,1)
end = datetime.datetime(2021,1,30)
```

- Choose a stock

```
aapl = web.DataReader("aapl", 'yahoo', start,end)
```

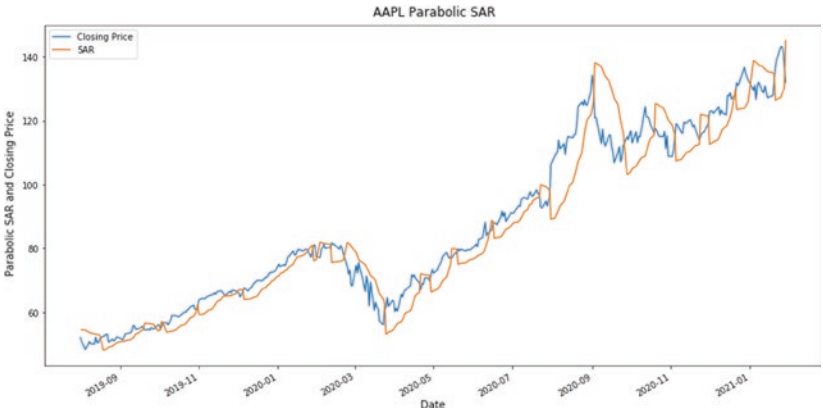


Fig. 30 AAPL Parabolic SAR

– Configure the parabolic SAR

```
aapl['SAR'] = talib.SAR(aapl['High'].values, aapl['Low'].values,
acceleration=0.02, maximum=0.2)
aapl
```

Date	*High*	*Low*	*Open*	*Close*	*Volume*	*Adj Close*	*SAR*
2019-08-01	54.507500	51.685001	53.474998	52.107498	216,071,600.0	51.311756	NaN
2019-08-02	51.607498	50.407501	51.382500	51.005001	163,448,400.0	50.226093	54.507500
2019-08-05	49.662498	48.145000	49.497501	48.334999	209,572,000.0	47.596859	54.425500
2019-08-06	49.517502	48.509998	49.077499	49.250000	143,299,200.0	48.497894	54.174280
2019-08-07	49.889999	48.455002	48.852501	49.759998	133,457,600.0	49.000103	53.933109

– Plot the Parabolic SAR with the closing price (Fig. 30).

```
aapl['Close'].plot(label='Closing Price',figsize=(16,8))
aapl['SAR'].plot(label= 'SAR')

_ = plt.xlabel('Date')

_ = plt.ylabel('Parabolic SAR and Closing Price')

_ = plt.title('AAPL Parabolic SAR')

plt.legend();
```

Table 1 Closing time in stock markets

Stock exchange	*Closing time*	*Time zone*
New York Stock Exchange (NYSE)	4:00 p.m	Eastern Time
London Stock Exchange (LSE)	4:30 p.m–4:35 p.m	Greenwich Mean Time
Bolsa de Madrid	5:30 p.m–5:45 p.m	Greenwich Mean Time + 1

Source Created by the author with information from (Bolsa de Madrid 2020) (London Stock Exchange Group 2020) (NYSE 2020)

The parabolic SAR is useful to buy or sell stocks depending on its relationship with the closing price of the stock, in this example, it is the stock of Apple. The parabolic SAR recommends us to buy if the SAR line is below the closing price and to sell if the SAR is above the closing price. As seen before, at the beginning of the chart, the SAR line and the closing price were very similar but as volatility has struck Apple, the opportunity for selling and buying is clearer. This also can be complemented with the stochastic oscillator.

Fast and Slow Stochastic Oscillators

Stochastic oscillators are momentum indicators, such as the SMA seen before or the SAR. The difference considering the stochastic oscillators is that it can be divided into *fast* and *slow,* and that it is considered usually during a period of fourteen days. The formula for calculating the fast stochastic oscillator, referred to as %K, is as follows:

Equation 4: Fast Stochastic Oscillator Equation

$$\%K = \frac{100 * \text{Closing price} - \text{Low price of the 14 previous trading sessions}}{(\text{Highest price in the last 14 day sessions} - \text{Low price of the 14 previous trading sessions}}. \tag{4}$$

The interpretation regarding $\%K$ is that if the result is 80 then the price is 8% above the prices in the last 14 days. The days can change based on the intuition and knowledge of the trader and it is usual to see a five-day period.

The *slow* indicator creates a change by applying a three(3)-day moving average to *fast* calculation (%*K*). The result is that the *slow* stochastic oscillator (%*D*) creates a signal line that is useful to know when to buy and when to sell. The process for callating a *fast* oscillator is as follow:

– Change the dates

```
start = datetime.datetime(2020,12,1)
end = datetime.datetime(2021,1,30)
```

It is important to create an oscillator with fewer data points than the one's used before. This is extremely useful for noticing the effect.

– Fast stochastic oscillator

```
aapl['fastk'], aapl['fastd'] = talib.STOCHF(aapl['High'].values,
aapl['Low'].values, aapl['Close'].values, fastk_period=14, fastd_period=3)
```

In the equation before, the standard fourteen days were used for the %*K* and three days for the %*D*.

– Slow stochastic oscillator

```
aapl['slowk'], aapl['slowd'] = talib.STOCH(aapl['High'].values,
aapl['Low'].values,  aapl['Close'].values, fastk_period=14,
slowk_period=3, slowd_period=3)
```

– Result

Date	*High*	*Low*	*Open*	*Close*	*Volume*	*Adj Close*	*SAR*	*slowk*	*slowd*	*fastk*	*fastd*
2021-01-25	145.089996	136.539993	143.070007	142.919998	157,611,700	142.919998	127.173966	87.180935	71.431681	88.401933	87.180935
2021-01-26	144.300003	141.369995	143.600006	143.160004	98,390,600	143.160004	128.248927	90.765333	85.085841	89.684699	90.765333
2021-01-27	144.300003	140.410004	143.429993	142.059998	140,843,800	142.059998	129.259392	87.155227	88.367165	83.379048	87.155227
2021-01-28	141.990005	136.699997	139.520004	137.089996	142,621,100	137.089996	130.209228	76.393343	84.771301	56.116282	76.393343
2021-01-29	136.740005	130.210007	135.830002	131.960007	177,180,600	131.960007	145.089996	55.823745	73.124105	27.975904	55.823745

– Plotting results (Figs. 31 and 32)

```
aapl['slowk'].plot(label='slowk',figsize=(16,8))
aapl['slowd'].plot(label= 'slowd')


_ = plt.xlabel('Date')

_ = plt.ylabel('Slow Stochastic Oscillator')

_ = plt.title('Slow Stochastic Oscillator - AAPL')

plt.legend();
```

```
aapl['fastk'].plot(label='fastk',figsize=(16,8))

aapl['fastd'].plot(label= 'fastd')

_ = plt.xlabel('Date')

_ = plt.ylabel('Fast Stochastic Oscillator')

_ = plt.title('Fast Stochastic Oscillator - AAPL')

plt.legend();
```

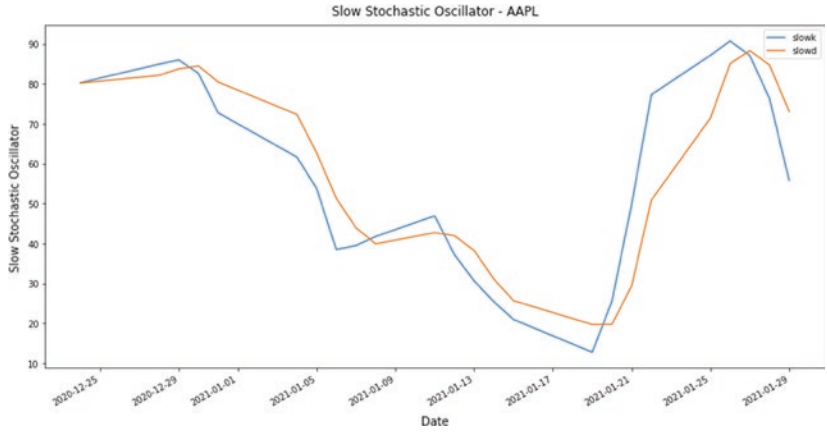


Fig. 31 Slow stochastic oscillator

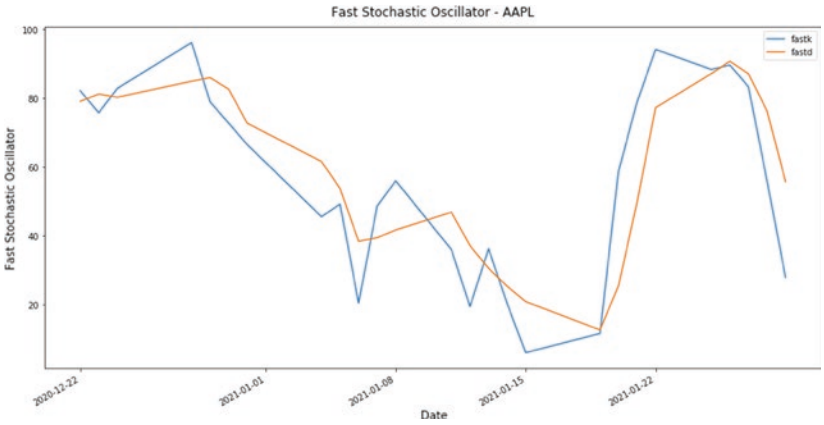


Fig. 32 Fast stochastic oscillator

References

Bang, Julie. 2019. *Candlestick bar*. Investopedia.

Basurto, Stefano. 2020. *Python trading toolbox: Introducing OHLC charts*, 7 January. Accessed March 31, 2020. https://towardsdatascience.com/trading-toolbox-03-ohlc-charts-95b48bb9d748.

Bollinger, John. 2018. *John Bollinger answers "What are Bollinger Bands?"*, n.d. Accessed March 2, 2020. https://www.bollingerbands.com/bollinger-bands.

Bolsa de Madrid. 2020. *Electronic Spanish Stock Market Interconnection System (SIBE)*, n.d. Accessed March 23, 2020. http://www.bolsamadrid.es/ing/Inversores/Agenda/HorarioMercado.aspx.

Chen, James. 2019. *Line Chart*, 12 August. Accessed March 25, 2020. https://www.investopedia.com/terms/l/linechart.asp.

Halton, Clay. 2019. *Line Graph*, 21 August. Accessed March 25, 2020. https://www.investopedia.com/terms/l/line-graph.asp.

Hayes, Adam. 2018. *Volume definition*, 4 February. Accessed March 25, 2020. https://www.investopedia.com/terms/v/volume.asp.

Hayes, Adam. 2020a. *Exponential Moving Average - EMA definition*, 8 July. Accessed April 2, 2020. https://www.investopedia.com/terms/e/ema.asp.

Hayes, Adam. 2020b. *Moving Average (MA)*, 31 March. Accessed March 31, 2020. https://www.investopedia.com/terms/m/movingaverage.asp.

Kevin, S. 2015. *Security analysis and portfolio management*. Delhi: PHI.

London Stock Exchange Group. 2020. *London Stock Exchange Group Business Day*, n.d. Accessed March 25, 2020. https://www.lseg.com/areas-expertise/

our-markets/london-stock-exchange/equities-markets/trading-services/business-days.
Mastromatteo, Davide. 2020. *Python args and kwargs: Demystified*. 09 September. Accessed March 31, 2020. https://realpython.com/python-kwargs-and-args/.
Milton, Adam. 2020. *Simple, exponential, and weighted moving averages*, 9 November. Accessed March 31, 2020. https://www.thebalance.com/simple-exponential-and-weighted-moving-averages-1031196.
Mitchell, Cory. 2019. *Understanding basic candlestick charts*. 19 December. Accessed March 30, 2020. https://www.investopedia.com/trading/candlestick-charting-what-is-it/.
Mitchell, Cory. 2020. *How to use volume to improve your trading*, 25 February. Accessed March 27, 2020. https://www.investopedia.com/articles/technical/02/010702.asp.
Murphy, Casey. 2020. *Investopedia*, 16 November. Accessed January 1, 2021. https://www.investopedia.com/trading/introduction-to-parabolic-sar/.
Murphy, John J. 1999. *Technical analysis of the financial markets*. New York: New York Institute of Finance.
NYSE. 2020. *TAQ closing prices*, n.d. Accessed March 25, 2020. https://www.nyse.com/market-data/historical/taq-nyse-closing-prices.
Posey, Luke. 2019. *Implementing MACD*, 30 March. Accessed April 2, 2020. https://towardsdatascience.com/implementing-macd-in-python-cc9b2280126a.

Valuation and Risk Models with Stocks

Abstract One of the most important aspects for the analysis of securities is to analyze its risk models and how to valuate the instrument. In the present chapter aims to explain the importance of risk, the different financial measures for understanding risk in a portfolio of securities and the impact on the returns of a portfolio.

Keywords Beta · Alpha · Risk · Valuation · Portfolio

This part of the book is centered on the management of risk. In this aspect, financial risk should be seen as the risk of the financial markets based on liquidity, operational and strategical risk. As seen before there are three tools for managing risk being fundamental analysis, technical analysis and quantitative analysis (Chen 2019).

Therefore, this section of the book is centered on Modern Portfolio Theory, Value at Risk and Monte Carlo Simulations. These three methods are imperative for understanding the financial markets and risk management.

Creating a Portfolio

Managing portfolios with multiple assets is one of the most interesting processes when working on finance. Given that there are different assets which can compose a portfolio, one of the most important processes of

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_7

establishing a portfolio is identifying its risk. For understanding how to build and analyze a portfolio this chapter will center on the process for creating, optimizing and evaluating a portfolio using Python.

When creating the portfolio there are different approaches that can be taken. There is always the portfolio that is created by the sentiment of the investor, the companies that they believe are the correct ones and the risk that he is willing to take. Considering risk, it depends if the investor is interested in bonds, cash, stocks or equivalents. In the case of this example, the most popular approach is to determine the market value (Bryant 2020).

The first step is to determine the companies that are performing in the index or choosing the companies that are of interest to the investor. For example, Disney, Netflix, Tesla and Amazon have been chosen. To the present time that the book was written the companies have been behaving greatly and creating strong returns.

- Creating a Portfolio with 4 variables

```
start = datetime.datetime(2014, 1, 1)
end = datetime.datetime(2019, 1, 1)

tickers = ('NFLX', 'DIS', 'TSLA', 'AMZN')
stocks = pd.DataFrame()
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo', start, end)['Close']
```

The second step then creating the portfolio is assigning the weights that the portfolio must have. For this approach, considering market capitalization is important.

- Determining market value

To determine market value two steps are essential. To know the last price of the security traded and the number of shares outstanding. For this, it is important to have a benchmark that can be used as a reference for the securities. Since the example establishes the last price as of December 26, 2018, then the prices can be obtained as follows:

```
stocks.tail()
```

Table 1 Shares outstanding of Tesla, Netflix, Amazon and Walt Disney

Company	*Shares Oustanding*[1]
Tesla Inc	172,721,000
Netflix Inc	436,599,000
Amazon.com, Inc	491,203,000
Walt Disney Company (The)	1,490,777,000

Source Elaborated by the author with information from Yahoo Finance and Nasdaq [a]Obtained from the https://www.nasdaq.com

Table 2 Market capitalization of Netflix, Tesla, Amazon and Walt Disney

Company	*Last Stock Price*	*Shares Oustanding*	*Market Capitalization*
Telsa Inc	326.09	172,721,000	56,322,590,890
Netflix Inc	253.67	436,599,000	110,752,068,330
Amazon.com, Inc	1470.90	491,203,000	722,510,492,700
Walt Disney Company (The)	105.83	1,490,777,000	157,768,929,910

Source Elaborated by the author with information from Yahoo Finance and Nasdaq

Date	*NFLX*	*DIS*	*TSLA*	*AMZN*
2018–12-26	253.669998	105.830002	326.089996	1470.900024
2018–12-27	255.570007	106.519997	316.130005	1461.640015
2018–12-28	256.079987	107.300003	333.869995	1478.020020

According to the NASDAQ, the shares outstanding of the company are as follows (Table 1):

To calculate the market capitalization of each company, the process is to multiply the last stock price with the shares outstanding (Table 2).

With the information, the next process is to obtain the market capitalization of the benchmark, in this case the Standard and Poor's 500 (S&P 500). The market capitalization for December 31, 2018 is 21.03 trillion (Table 3).

With the information above it is easier to understand the market value of the stock and the relation between the benchmark and the stock that has been chosen for the portfolio. In this case, the most significant stock in the portfolio is Amazon with a 69.98%.

For example of the portfolio, it may be important to add other companies that can be included to weigh down the participation of Amazon, but for this example the assigning of weights is as follows:

[1] Obtained from the https://www.nasdaq.com

Table 3 Market capitalization and portfolio weight

Company	*Market Capitalization*	*Portfolio Weight of the S&P 500*
Telsa Inc	56,322,590,890	5.38%
Netflix Inc	110,752,068,330	10.57%
Amazon.com, Inc	722,510,492,700	68.98%
Walt Disney Company (The)	157,768,929,910	15.06%
Total	1,047,354,081,830	100%

Source Elaborated by the author with information from Yahoo Finance and Nasdaq

- Calculate stocks returns

```
stocks_return=(stocks/stocks.shift(1)
```

- Calculate the weights of the portfolio based on market capitalization

```
portfolio_weights = np.array([0.0538, 0.1057, 0.6898, 0.1506])
```

- Create a weighted return portfolio

```
weighted_returns_portfolio=stocks_return.mul(portfolio_weights, axis= 1)
weighted_returns_portfolio.head().dropna()
```

- Create a stock portfolio based on the weights assigned

```
stocks_return['Portfolio' ]=weighted_returns_portfolio.sum(axis= 1 ).dropna()
```

Calculating Statistical Measures on a Portfolio

Now that the portfolio is created there are certain statistical measures that can be calculated. The first one is the portfolio standard deviation which is as follows:

Equation 1: Portfolio standard deviation

$$\sigma_{Portfolio} = \sqrt{w_t * \Sigma * w}$$

w_t= transposed portfolio of the weights assigned
Σ = the covariance matrix of the returns
w = weights

The first step is to obtain the annual covariance. For the annual covariance the number of days that will be used is 252, a more conventional approach to the 360 or 365. The days are based on the stock market available days:

- Calculate the covariance of the stocks and the portfolio

```
stocks_covariance = stocks_return.cov()
```

Covariance Matrix of the Stock and the Portfolio

	NFLX	*DIS*	*TSLA*	*AMZN*	*Portfolio*
NFLX	0.000735	0.000089	0.000247	0.000236	0.000255
DIS	0.000089	0.000142	0.000090	0.000080	0.000094
TSLA	0.000247	0.000090	0.000797	0.000186	0.000601
AMZN	0.000236	0.000080	0.000186	0.000381	0.000207
Portfolio	0.000255	0.000094	0.000601	0.000207	0.001265

- Annualize the covariance

```
annualized_covariance = stocks_covariance * 252
annualized_covariance
```

Annualized Covariance of the Stocks and the Portfolio

	NFLX	*DIS*	*TSLA*	*AMZN*	*Portfolio*
NFLX	0.185274	0.022497	0.062368	0.059461	0.064322
DIS	0.022497	0.035697	0.022763	0.020037	0.023703
TSLA	0.062368	0.022763	0.200928	0.046951	0.151433
AMZN	0.059461	0.020037	0.046951	0.096091	0.052175
Portfolio	0.064322	0.023703	0.151433	0.052175	0.318702

- Obtain the standard deviation of the portfolio

```
portfolio_sd = np.sqrt(np.dot(portfolio_weights.T,  np.dot[2](annualized_
covariance, portfolio_weights)))
portfolio_sd
```

```
0.34392103687572534
```

[2] The np.dot converts the two data sets into one. For more information visit: https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html.

The volatility of the portfolio is 34.49% with can be considered as a highly volatile portfolio. This is important because when analyzing the beta, the behavior of the market and the stock will be included into the analysis.

The Capital Asset Pricing Model

The Capital Asset Pricing Model better known as CAPM is a model created by William Sharpe based on the work of Harry Markowitz. Its major assumptions are that the offer of financial assets is equal to the demand of financial assets (Mullins 1982). When the assumption is analyzed under perfect competitiveness, which means that the price is determined by the offer and demand of economic agents, hence determine the price of the asset.

Another important aspect of the CAPM is that it only considers the systematic risk (market risk). Systematic risk can be understood as the risk that is not diversifiable and therefore cannot be reduced (Fontinelle 2019). Risk, as a whole is divided as follows:

Equation 2: Total risk

$$\textit{Total Risk} = \textit{systematic risk} + \textit{non systematic risk}$$

The Beta

The measurement of the systematic risk is through the *beta*, which is a degree of sensitivity that includes the variation of an asset compared with an index that is used as a benchmark.

Equation 3: Beta calculation with correlation

$$\beta_i = \rho_{im} \frac{\sigma_i}{\sigma_m}$$

ρ_{im} = Correlation between asset i and the market
σ_i = variance of asset i
σ_m = variance of the *market*

Another way of calculating the *beta* is through the following formula:

Equation 4: Beta calculation with covariance

$$\beta_i = \frac{cov_{(i,m)}}{\sigma_m}$$

$cov_{(i,m)}$= covariance of assets i and *market*
σ_m= variance of the *market*

Given that the covariance will be obtained for calculating the *beta*, *it* is important to elaborate a covariance matrix. First, the process will be explained with the covariance formula and then it will be integrated with the correlation formula.

Before coding in Python there are certain aspects that should be clarified to understand the process of computing the *beta* with covariance:

- Using a *for loop*: The program below uses a simple *for loop* because there is a need here to create a DataFrame with the two variables. The process is simple, and for more information there are different ways a for loop[3] can be used. See *For loop*.
- Choosing a market for comparison: For example, the S&P 500 (^GSPC) was used because it is an interesting market reference when trying to determine the behavior of the company.
- Using iloc: iloc[4] is a *pandas* DataFrame function that selects a position. It can be used to slice the DataFrame or to create new columns. Please visit the chapter titled *The Basics* for more information.
- Using 252 days: The concept of 252 days comes from the regular trading hours (RTH). It is a rule of thumb (Mitra y Mitra 2011). The covariance and the variance are multiplied by this to convert it into daily.

These aspects are important since they will be repeated throughout the following sections of the book.

[3] An excellent resource for understanding for loops can be found here: https://www.w3schools.com/python/python_for_loops.asp.

[4] For more information on using iloc and examples visit: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html.

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Setting time and date

- start = datetime.datetime(2014, 1, 1)
- end = datetime.datetime(2019, 1, 1 2019, 1, 1)

- Using a *for loop* for obtaining tickers

```
tickers = ['DIS', '^GSPC' ]
stocks = pd.DataFrame()
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo' , start, end)['Close' ]
```

- Creating logarithmic returns

```
stocks_return=np.log(stocks/stocks.shift(1 ))
```

- Calculating the covariance matrix

```
covariance=stocks_return.cov() * 252
covariance
```

Covariance Matrix

	DIS	*^GSPS*
DIS	0.035899	0.015926
^GSPS	0.015926	0.017540

- Setting the market covariance into a variable

```
covariance_market = covariance.iloc[0,1 ]
covariance_market
```

- Calculating the variance of the market

```
market_variance = stocks_return['^GSPC' ].var( ) * 252
market_variance
```

- Calculating Beta with covariance

```
beta_Disney = covariance_market/market_variance
beta_Disney
```

```
Beta = 0.90796359355848766
```

For calculating the *beta* with correlation, the formula, it is easier but it is less common. It is important to remember that when comparing *beta*, the same days of data are being used, and that this is applied to all the variables. There are different approaches to the *beta* but the most notable aspect is to understand the result. To.

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader  import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Setting time and date

```
start = datetime.datetime(2014, 1, 1 )
end = datetime.datetime(2019, 1, 1 )
```

- Using a *for loop* for obtaining tickers

```
tickers = [NFLX, '^GSPC' ]
stocks = pd.DataFrame()
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo' , start, end)[ 'Close' ]
```

- Creating logarithmic returns

```
stocks_return=np.log(stocks/stocks.shift(1 ))
```

- Calculating correlation

```
correlation=stocks_return.corr()
correlation
```

Correlation Matrix

	DIS	*NFLX*
NFLX	1.000000	0.467229
^GSPS	0.467229	1.000000

- Inserting Correlation into a variable

```
correlation_NFLX_GSPC=correlation.iloc[0,1 ]
correlation_NFLX_GSPC
```

- Netflix variance

```
variance_NFLX=stocks_return['NFLX' ].var()
variance_NFLX
```

- Market variance

```
variance_GSPC=stocks_return['^GSPC' ].var()
variance_GSPC
```

- Beta

```
beta = correlation_NFLX_GSPC * (variance_NFLX/variance_NFLX)
beta
```

```
Beta = 0.46722863704340151
```

The beta above is annualized since the correlation is obtained by using all the data and comparing it. This is why the first formula is used more often because it is capable of representing the beta using daily information. Notice that correlation can change if the data is used daily, monthly or annually and this could lead to results that are not comparable.

The next step is to interpret the *beta coefficient*. The interpretation of the *beta coefficient* is very useful since it helps understand the relation between a security and the market it trades. This is fundamental when analyzing a portfolio because it is imperative to comprehend what happens when the market moves with a certain security that is included in the portfolio (Table 4).

The *beta coefficients* results*:*

```
Beta Disney = 0.9079
Beta Netflix = 0.4672
```

Both of them are less volatile than the market, meaning that if the S&P 500 shifts in an upward direction by 1% then Disney will shift by 0.91% and Netflix by 0.46%. The beta is important for analyzing the process in which the assets behave when compared to a market.

Table 4 The Beta Table

$\beta_i = 1$	*The asset is exactly as volatile as the market it is compared with*
$\beta_i > 1$	The asset is more volatile than the market it is compared with
$\beta_i < 1 > 0$	The asset is less volatile than the market it is compared with
$\beta_i = 0$	The asset is not correlated with the market it is compared with
$\beta_i < 0$	Negatively correlated with the market it is compared with

The Beta and the CAPM

The CAPM is considered part of the factor analysis because it allows the understanding of the relationship between variables. It is important to consider that when using the factor analysis, the key aspect understanding the relation of the correlated variables with the factors.

The first step for developing the CAPM is calculate the excess returns. An excess return happens when the asset or portfolio exceeds the risk-free return. The equation is as follows:

Equation 5: Excess return

$$Excess\ Return = Return - Risk\ Free\ Return$$

A risk-free return can be defined as an investment that with zero risk (lowest risk possible) guarantees a return. Usually for the risk-free rate the U.S. Treasury Bills (known as T-Bills) are considered because they are backed by the U.S. Government and the risk of default is minimum.

To understand the relation between the Risk-Free Return, the Beta and the CAPM the easiest way is to understand the model by its equation:

Equation 6: Capital Asset Pricing Model

$$E(R_p) - RF = \beta_p(E(R_m) - RF)$$

$E(R_p) - RF$ = The excess in the expected return of Portfolio P
$E(R_m) - RF$ = The excess expected return of market portfolio
RF = Risk-Free Return
β_p = Beta of the portfolio

According to the equation, the first step for calculating the CAPM is to create a portfolio. To create a portfolio, one must understand that it is composed of different assets and that those assets should have a different weight.

The weights are known as portfolio weights and there are different methods for calculating the weights but the example will focus on the market value method seen before.

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Creating a Portfolio with 4 variables

```
start = datetime.datetime(2014, 1, 1 )
end = datetime.datetime(2019, 1, 1 )
tickers = ['NFLX', 'DIS', 'TSLA', 'AMZN' ]
stocks = pd.DataFrame()for x in tickers:
stocks[x]=web.DataReader(x, 'yahoo', start, end)['Close']
```

The second step when building a portfolio is identifying the weights that the model will consider. For this it is important to know how much money is going to be invested in the model. For this example, the investment will be of USD 500,000 and it will be distributed according to the following information (Table 5):

Considering the above, the subsequent step is to be familiar with the construction of the portfolio based on the company that is going to be chosen. One of the key aspects is that the sum of the portfolio weights

Table 5 Investing USD 500,000

Company	*Investment*	*Portfolio Weight*
Telsa Inc	130,000	26%
Netflix Inc	100,000	20%
Amazon.com, Inc	170,000	34%
Walt Disney Company (The)	100,000	20%
Total	500,000	100%

has to add to a 100% or to 1 depending on if you are using decimal or percentage. This is important because of the portfolio return formula given below:

Equation 7: Portfolio Return

$$R_p = R_{a1}w_{a1} + R_{a2}w_{a2} + R_{a3}w_{a3} + \ldots + R_{an}w_{ax}$$

R_p = Return of the portfolio
R_{a1} = Return of the first asset
w_{a1} = Weight of the first asset.

- Weights for the portfolio

```
portfolio_weights = np.array([0.30, 0.20, 0.25, 0.25 ])
```

- Calculating return for each company on the portfolio

```
portfolio_weights = np.array([0.30, 0.20, 0.25, 0.25 ])
```

- Creating a portfolio with four companies using close price for a time series between January 1, 2014 to January 1, 2019

```
start = datetime.datetime(2014, 1, 1 )
end = datetime.datetime(2019, 1, 1 )

tickers = ['NFLX', 'DIS', 'TSLA', 'AMZN' ]
stocks = pd.DataFrame()
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo' , start, end)[ 'Close' ]
```

- Create logarithmic returns in the portfolio

```
stocks_return=np.log(stocks/stocks.shift(1 ))
```

- Drop missing values in the series and visualize the data

```
stocks_return.dropna().head()
```

- Calculate the weighted stock returns of the portfolio

```
weighted_returns_portfolio = stocks_return.mul(portfolio_weights, axis = 1 )
```

- Calculating the returns of a portfolio

```
stocks_return['Portfolio' ] = weighted_returns_portfolio.sum(axis = 1 )
```

- Calculate the cumulative returns of the portfolio and plot them

```
cumulative_returns_porfolio = ((1 + stocks_return['Portfolio' ]).
cumprod()-1).

cumulative_returns_porfolio.plot(label = 'Cumulative Returns of the Portfolio' ,fig-
size = (19,8),title = 'Cumulative Returns')

_ = plt.xlabel('Date' )

_ = plt.ylabel('Portfolio Cumulative Return' )
```

The example above is a perfect example for understanding the behavior of a portfolio considering four companies. As the plot above demonstrates, the portfolio went from a cumulative return of approximately 0.4 in 2015 to above 2.0 in 2018. The last part of the graph exemplifies an interesting fall considering the portfolio (Fig. 1).

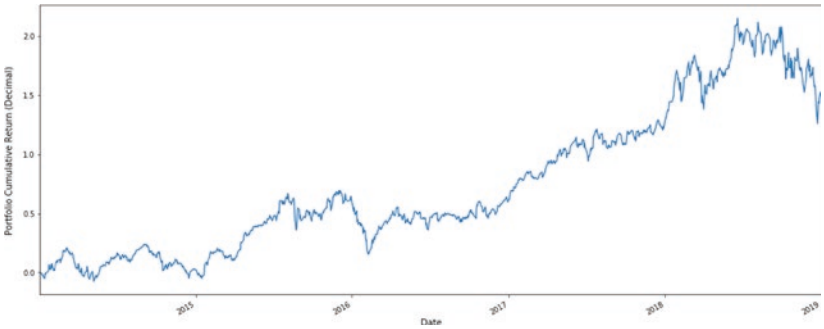


Fig. 1 Cumulative returns of the portfolio (*Source* Elaborated by the author with information from Yahoo Finance)

As the CAPM model specifies, it is important to calculate the excess based on the difference of the Risk-Free Rate and the Portfolio Returns. To choose the risk-free rate concerning a T-Bill should be based on the duration of the investment. For example, if the portfolio that is being built is aimed at 10 years then T-Bill that is appropriate is the 10-year T-Bill.

To calculate the real risk-free rate the process is as follows:

Equation 8: Real Risk

$$Real\ Risk - Free\ Rate = (Risk\ Free - Rate - Inflation)$$

At the time consulted for the present chapter, the T-Bill yield for 10 years was 2.53% and the inflation was 1.5%. The result of the real risk-free rate is 1.03%. If the model is using the real risk-free rate the column shall be added:

- Add the real risk-free rate column

```
stocks_return['RF Rate' ]=0.0103
```

With the column of the real risk-free rate and the portfolio returns, the excess can be calculated very easy:

```
stocks_return['excess' ]=stocks_return['Portfolio' ] - stocks_return['RF Rate' ]
```

With the information, the only variable missing for the CAPM is the beta. For this next step, the SPY (S&P 500) will be used:

- Calculating returns for the SPY index

```
start = datetime.datetime(2014, 1, 2 )
end = datetime.datetime(2019, 1, 1 )

stocks_return['Market' ]=web.DataReader('SPY','yahoo' ,start,end)[ 'Close' ]

stocks_return['Market' ]=(stocks_return['Market' ]/stocks_return['Market' ].
shift(1 ))
stocks_return.head().dropna()
```

The second step is to calculate the excess return of the market compared with the risk-free rate. This is basic because it standardizes the process by which the market behaves compared to the risk-free rate and the portfolio.

- Calculating the excess return of the market

```
stocks_return['excess market' ]=stocks_return['Market' ] - stocks_return['RF Rate' ]
stocks_return.head().dropna()
```

To obtain the beta there are different processes, for this example the process of the beta equation using covariance and variance will be used. The equation is as follows:

Equation 9: Beta calculation with covariance

$$\beta_p = \frac{Covar(R_p, R_B)}{Var(R_B)}$$

β_p = Beta of the portfolio
R_p = Return of the portfolio
R_B = Return of the benchmark

The next step is to obtain the covariance matrix. From the covariance matrix the coefficient can be inserted into a variable:

- Covariance Matrix

```
covariance_matrix=stocks_return[['excess', 'excess market' ]].cov()
covariance_matrix
```

Covariance Matrix

	excess	*excess market*
excess	0.001065	0.000091
excess market	0.000091	0.000070

- Covariance Coefficient

```
covariance_coefficient = covariance_coefficient = covariance_matrix.
iloc[0, 1]
```

Once the covariance is obtained the next process is to insert the variance of the portfolio into the process.

- Calculating the variance

```
variance_coefficient = stocks_return['excess market' ].var()
variance_coefficient
```

With the variance and the covariance, the beta can be obtained by using the formula described before.

- Calculating Beta

```
beta = covariance_coefficient/variance_coefficient
beta
```

```
1.2979841588534764
```

Considering the process, the beta demonstrates that the portfolio is more volatile than the market. Another interpretation is that the portfolio is 29% more volatile than the S&P 500 or that for every 1% of movement in the market there will be a 1.29% of rise or fall in the portfolio.

Sharpe Ratio

The Sharpe Ratio was created by William F. Sharpe based on the importance of understanding the relation between risk and returns. The Sharpe Ratio is considered one of the rentability and risk ratios.

When the Sharpe Ratio is higher the better it is considered since the denominator is standard deviation or risk. It is useful when comparing peers, for example in an exchange-traded fund (ETF) (Hargrave 2019).

Equation 10: Sharpe Ratio

$$Sharpe\ Ratio = \frac{R_p - R_f}{\sigma_p}$$

where:
R_p = returns of the portfolio
R_f = risk-free rate
σ_p = standard deviation of the portfolio excess returns.

For obtaining the Sharpe Ratio the first step is to create the portfolio using the method that has been used before:

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Creating the portfolio based on an end and start date

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

tickers = ['F','FCAU', 'TM' ,]
stocks = pd.DataFrame( )
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo' , start, end)[ 'Close' ]
```

In the example above the companies that are being used are Ford (F), Fiat-Chrysler (FCAU) and Toyota Motors (TM). The dates are from January 1, 2018 to April 1, 2019. For this example, a for loop was used

as an easiest way to understand the process by which the stocks can be added.

- Analyze correlation between variables

```
stocks.corr()
```

Correlation between companies

	F	*FCAU*	*TM*
F	1.000000	0.873512	0.837771
FCAU	0.873512	1.000000	0.852925
TM	0.837771	0.852925	1.000000

Concerning the correlation, the portfolio will be strongly correlated between the securities. This is often a recommendation to diversify the portfolio and to add companies that are in a different industry.

- Choose weights for the portfolio

```
portfolio_weights = np.array([0.33, 0.33, 0.34 ])
```

- Returns of the stocks using percent change

```
stocks_return=stocks.pct_change(1 ).dropna()
```

The use of the percent change is an easier approach because it takes the last return and calculates the change in percentages, which is another way of calculating the stocks returns without using logarithmic returns. Mathematically it is not recommended to use logarithmic model to calculate returns (Fig. 2).

- Multiply the returns of the portfolio with each stock's weight

```
weighted_returns_portfolio=stocks_return.mul(portfolio_weights, axis=1 )
```

- Create a variable for the portfolio by calculating the sum of the returns

```
stocks_return['Portfolio' ]=weighted_returns_portfolio.sum(axis=1 ).
dropna()
stocks_return.tail()
```

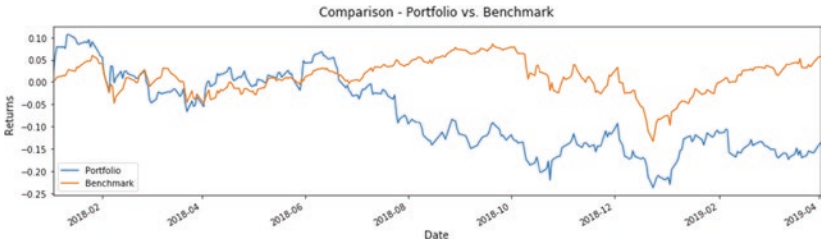


Fig. 2 Comparing Benchmark and Portfolio (*Source* Elaborated by the author with information from Yahoo Finance)

- Add the benchmark by using the S&P 500 and calculating the returns

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

stocks_return['Benchmark' ]=web.DataReader('SPY','yahoo' ,start,end)[ 'Close' ]
stocks_return['Benchmark' ] = stocks_return['Benchmark' ].pct_change(1).
dropna()
stocks_return.dropna().tail()
```

- Obtaining the cumulative returns and plotting the portfolio

```
CumulativeReturns = ((1 + stocks_return[['Portfolio','Benchmark' ]]).cum-
prod()-1 )
CumulativeReturns.plot(figsize=(16,4 ))
_=plt.ylabel('Returns' )
_=plt.title('Comparison - Portfolio vs. Benchmark' )
_=plt.xlabel('Date' )
plt.show()
```

- Create a scatterplot to identify the correlation between the portfolio and the benchmark

```
plt.scatter(stocks_return['Portfolio' ],stocks_return['Benchmark' ],alpha=0.80 );
_=plt.ylabel('Returns' )
_=plt.title('Correlation - Portfolio vs. Benchmark' )
_=plt.xlabel('Date' )
```

- Create a new DataFrame for the portfolio and the benchmark for calculating the correlation

```
portfolio_benchmark = pd.concat([stocks_return['Portfolio'],stocks_return['Benchmark']],axis=1).dropna()
portfolio_benchmark.columns = ['Portfolio',' Benchmark']
```

- Obtain the correlation between the portfolio and the benchmark

```
correlation = portfolio_benchmark.corr()
correlation
```

Correlation Between Portfolio and the Benchmark

	Portfolio	*Benchmark*
Portfolio	1.000000	0.666916
Benchmark	0.666916	1.000000

- Add risk-free rate based on the benchmark of Treasury Bills chosen

```
stocks_return['RF Rate'] = 0.0103.
```

- Calculate the excess of the portfolio

```
stocks_return['excess'] = stocks_return['Portfolio'] - stocks_return['RF Rate']
```

- Calculate the excess of the benchmark

```
stocks_return['excess_b'] = stocks_return['Benchmark'] - stocks_return['RF Rate']
```

- Calculating the Sharpe Ratio

```
sharpe_ratio = ((stocks_return['Portfolio'].mean() - stocks_return['RF Rate'].mean()))/stocks_return['Portfolio'].std()
sharpe_ratio
```

```
-0.7134969693218578
```

- Calculating the annual Sharpe Ratio

```
import math
annual_days = 252

sharpe_ratio_annual = sharpe_ratio * math.sqrt(annual_days)
sharpe_ratio_annual
```

```
-11.18589313516733
```

Considering that the results are negative, the problem lies that the mean return of the portfolio is smaller than the risk-free rate. As discussed before, the return of the portfolio is lower than the risk-free rate exemplifies that the portfolio is not performing above our lowest target (the RF Rate) and therefore a portfolio is not effective (Fig. 3).

By using highly correlated assets as a portfolio, returns were sacrificed and the risk was higher because if there is a fall in one security the other companies will respond in the same way. This is an important lesson when building a portfolio and not considering the warnings when analyzing correlation.

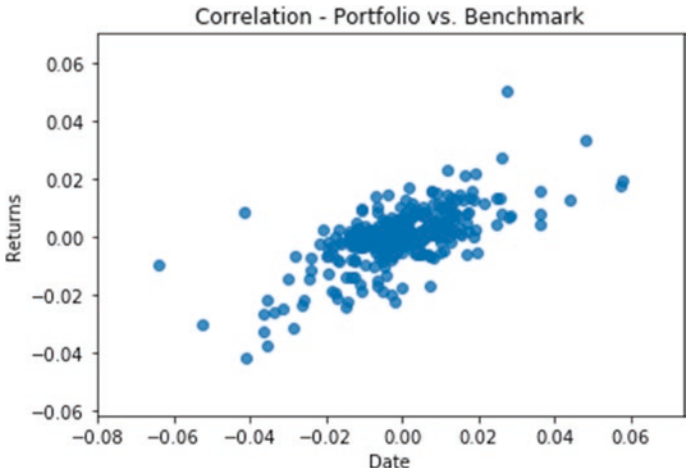


Fig. 3 Correlation plot between Portfolio and Benchmark (*Source* Elaborated by the author with information from Yahoo Finance)

Traynor Ratio

The ratio was created by John Traynor in 1965 and measure the rentability compared to risk. The rule of thumb of the ratio is that an indicator that is higher is a result of the portfolio management. When analyzing the Traynor Ratio, if it is negative, the portfolio has underperformed the risk-free rate.

Since the Traynor Ratio measures returns in excess earned on a riskless investment compared by using a per market risk, it is useful because it compares the returns to the risk of the investor (Keaton 2020). As seen in the following equation, the main difference between a Sharpe Ratio and a Traynor Ratio is the use of beta.

Equation 11: Traynor Ratio

$$Traynor\ Ratio = \frac{R_p - R_f}{\beta_p}$$

where:
R_p = returns of the portfolio
R_f = risk-free rate
β_p = Beta of the portfolio

For example, the data set that was used for the Sharpe Ratio will be used. This is important for comparison between the ratios.

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader  import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Creating the portfolio based on an end and start date

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )
tickers = ['F','FCAU', 'TM' ,]
stocks = pd.DataFrame()
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo' , start, end)[ 'Close' ]
```

- Choose weights for the portfolio

```
portfolio_weights = np.array([0.33, 0.33, 0.34 ])
```

- Returns of the stocks using percent change

```
stocks_return=stocks.pct_change(1 ).dropna()
```

- Multiply the returns of the portfolio with each stock's weight

```
weighted_returns_portfolio=stocks_return.mul(portfolio_weights,
axis=1 )
```

- Create a variable for the portfolio by calculating the sum of the returns

```
stocks_return['Portfolio' ]=weighted_returns_portfolio.sum(axis=1 ).
dropna()
stocks_return.tail()
```

- Add the benchmark by using the S&P 500 and calculating the returns

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

stocks_return['Benchmark' ]=web.DataReader('SPY','yahoo' ,start,end)[ 'Close' ]
stocks_return['Benchmark' ]=stocks_return['Benchmark' ].pct_change(1).
dropna()
stocks_return.dropna().tail()
```

- Obtaining the cumulative returns and plotting the portfolio

```
CumulativeReturns = ((1 + stocks_return[['Portfolio','Benchmark' ]]).cum-
prod()-1 )
CumulativeReturns.plot(figsize = (16,4 ))
_ = plt.ylabel('Returns' )
_ = plt.title('Comparison - Portfolio vs. Benchmark' )
_ = plt.xlabel('Date' )
plt.show()
```

- Add risk-free rate based on the benchmark of Treasury Bills chosen

```
stocks_return['RF Rate' ] = 0.0103
```

- Calculate the covariance of the stocks

```
covariance = stocks_return.cov() * 252
covariance
```

	F	*FCAU*	*TM*	*Portfolio*	*Benchmark*
F	0.081996	0.056534	0.024013	0.053879	0.023830
FCAU	0.056534	0.167189	0.036997	0.086408	0.035823
TM	0.024013	0.036997	0.035576	0.032229	0.019736
Portfolio	0.053879	0.086408	0.032229	0.057070	0.026396
Benchmark	0.023830	0.035823	0.019736	0.026396	0.027260

- Calculate the covariance between the market and the portfolio

```
covariance_market = covariance.iloc[3,4 ]
covariance_market
```

```
0.026395748767295644
```

- Calculate the variance of the benchmark

```
market_variance = stocks_return['Benchmark' ].var() * 252
market_variance
```

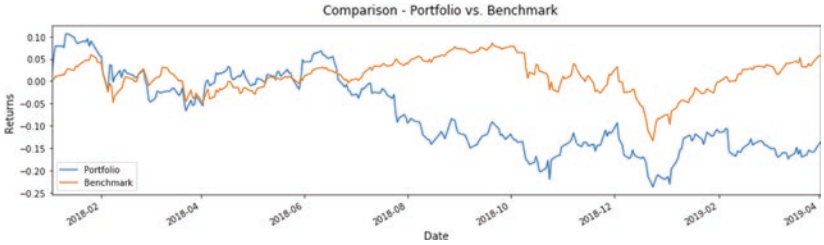


Fig. 4 Comparsion between portfolio and benchmark (*Source* Elaborated by the author with information from Yahoo Finance)

- Calculate the beta of the portfolio

```
portfolio_beta = covariance_market / market_variance
portfolio_beta
```

```
0.96828357625053052
```

```
traynor_ratio = ((stocks_return['Portfolio'].mean() - stocks_return['RF Rate' ].
mean()))/portfolio_beta
traynor_ratio
```

```
-0.011269427871314531
```

As a result, the Traynor Ratio is negative, which means that the portfolio is not performing better than the risk-free rate. The result of the Sharpe Ratio was negative also, so it complements the information given by Traynor. The main difference between the Sharpe and the Traynor ratio is that it compares with the beta and not the volatility (Fig. 4).

Jensen's Measure

The Jensen's Measure also known as alpha was created in 1968 with the purpose of measuring the relationship between the return of the portfolio in comparison with another portfolio return with the same risk, same reference market and under the same parameters (Chen 2019).

Equation 12: Jensen's measure

$$Alpha\ (\alpha) = R_p - (R_f + \beta_p(R_m - R_p)$$

R_p = returns of the portfolio
R_f = risk-free rate
β_p = Beta of the portfolio
R_m = return of the market.

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Creating the portfolio based on an end and start date

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

tickers = ['F','FCAU', 'TM',]
stocks = pd.DataFrame( )
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo' , start, end)[ 'Close' ]
```

- Choose weights for the portfolio

```
portfolio_weights = np.array([0.33, 0.33, 0.34 ])
```

- Returns of the stocks using percent change

```
stocks_return=stocks.pct_change(1 ).dropna( )
```

- Multiply the returns of the portfolio with each stock's weight

```
weighted_returns_portfolio = stocks_return.mul(portfolio_weights,
axis = 1 )
```

- Create a variable for the portfolio by calculating the sum of the returns

```
stocks_return['Portfolio' ] = weighted_returns_portfolio.sum(axis = 1 ).
dropna()
stocks_return.tail()
```

- Add the benchmark by using the S&P 500 and calculating the returns

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

stocks_return['Benchmark' ] = web.DataReader('SPY','yahoo' ,start,end)[ 'Close' ]
stocks_return['Benchmark' ] = stocks_return['Benchmark' ].pct_change(1).
dropna()
stocks_return.dropna().tail()
```

- Add risk-free rate based on the benchmark of Treasury Bills chosen

```
stocks_return['RF Rate' ] = 0.0103
```

- Calculate the covariance of the stocks

```
covariance = stocks_return.cov() * 252
covariance
```

	F	*FCAU*	*TM*	*Portfolio*	*Benchmark*
F	0.081996	0.056534	0.024013	0.053879	0.023830
FCAU	0.056534	0.167189	0.036997	0.086408	0.035823
TM	0.024013	0.036997	0.035576	0.032229	0.019736
Portfolio	0.053879	0.086408	0.032229	0.057070	0.026396
Benchmark	0.023830	0.035823	0.019736	0.026396	0.027260

- Choose the covariance between the market and the portfolio

```
covariance_market = covariance.iloc[3,4 ]
covariance_market
```

```
0.026395748767295644
```

- Calculate the variance of the benchmark

```
market_variance = stocks_return['Benchmark' ].var() * 252
market_variance
```

- Calculate the beta of the portfolio

```
portfolio_beta = covariance_market / market_variance
portfolio_beta
```

```
0.96828357625053052
```

- Calculate the return of the portfolio

```
portfolio_return = stocks_return['Portfolio' ].mean()
```

- Calculate the risk-free rate

```
risk_free_rate = stocks_return['RF Rate' ].mean()
```

- Calculate the Alpha

```
alpha = portfolio_return - (risk_free_rate + portfolio_beta *(portfolio_
return - risk_free_rate)).
alpha
```

```
-0.00034608967689839253
```

The alpha is negative and so is the Traynor and the Sharpe Ratio. This is a perfect example of a portfolio that is performing negatively when compared to the risk-free rate. For decision making with alpha the following table is important (Table 6):

Table 6 Alpha decision making

$(\alpha) > 0$	the portfolio has gained value
$(\alpha) < 0$	the portfolio has lost value

Information Ratio

The information ratio analyzes the excess of the return when comparing the portfolio without risk with the one supported by the investment. The effect that the ratio measures is how the portfolio deviates from the benchmark (Murphy 2019) The name is based on the consideration that the manager of the portfolio has *special information* and therefore, he will out beat the benchmark. The formula is as follows:

$$Information\ Ratio = \frac{(R_p - R_m)}{TE}$$

TE = standard deviation of the difference between the portfolio and the benchmark
R_p = Porfolio Return
β_p = Beta of the portfolio
R_m = return of the market

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Creating the portfolio based on an end and start date

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

tickers = ['F','FCAU', 'TM',]
stocks = pd.DataFrame()
for x in tickers:
    stocks[x]=web.DataReader(x, 'yahoo', start, end)[ 'Close' ]
```

- Choose weights for the portfolio

```
portfolio_weights = np.array([0.33, 0.33, 0.34 ])
```

Returns of the stocks using percent change

```
stocks_return=stocks.pct_change(1 ).dropna()
```

- Multiply the returns of the portfolio with each stock's weight

```
weighted_returns_portfolio=stocks_return.mul(portfolio_weights,
axis=1 )
```

- Create a variable for the portfolio by calculating the sum of the returns.

```
stocks_return[ 'Portfolio' ]=weighted_returns_portfolio.sum(axis=1 ).
dropna()
stocks_return.tail()
```

- Add the benchmark by using the S&P 500 and calculating the returns

```
start = datetime.datetime(2018, 1, 2 )
end = datetime.datetime(2019, 4, 1 )

stocks_return['Benchmark' ]=web.DataReader('SPY','yahoo' ,start,end)['Close' ]
stocks_return[ 'Benchmark' ]=stocks_return[ 'Benchmark' ].pct_change(1).
dropna()
stocks_return.dropna().tail()
```

- Add risk-free rate based on the benchmark of Treasury Bills chosen

```
stocks_return['RF Rate' ]=0.0103
```

- Calculate the covariance of the stocks

```
covariance=stocks_return.cov() * 252
covariance
```

	F	*FCAU*	*TM*	*Portfolio*	*Benchmark*
F	0.081996	0.056534	0.024013	0.053879	0.023830
FCAU	0.056534	0.167189	0.036997	0.086408	0.035823
TM	0.024013	0.036997	0.035576	0.032229	0.019736
Portfolio	0.053879	0.086408	0.032229	0.057070	0.026396
Benchmark	0.023830	0.035823	0.019736	0.026396	0.027260

- Choose the covariance between the market and the portfolio

```
covariance_market=covariance.iloc[3,4 ]
covariance_market
```

```
0.026395748767295644
```

- Calculate the variance of the benchmark

```
market_variance=stocks_return['Benchmark' ].var() * 252
market_variance
```

- Calculate the beta of the portfolio

```
portfolio_beta=covariance_market / market_variance
portfolio_beta
```

```
0.96828357625053052
```

- Calculate the return of the portfolio

```
portfolio_return=stocks_return['Portfolio' ].mean()
```

- Calculate the return of the benchmark

```
portfolio_return = stocks_return['Benchmark' ].mean()
```

- Difference between the return of the portfolio and the benchmark

```
difference_benchmark_portfolio = stocks_return['Portfolio'] - stocks_
return['Benchmark' ]
```

- Calculate the tracking error

```
tracking_error = difference_benchmark_portfolio.std()
```

Calculate the information

```
information_ratio = ((portfolio_return - benchmark_return))/ tracking_error
information_ratio
```

```
-0.07703212677308867
```

The result demonstrates that there is no excess between the returns of the portfolio and the benchmark leading to infer that the benchmark has outperformed the portfolio. As a conclusion concerning the ratios, the portfolio has performed worse than the risk-free rate and the benchmark.

- Applying the knowledge to an investment

The following exercise is based on an investment of USD 300,000 in the following securities with a specific allocation:

- Amazon 30% allocation—USD 90,0000
- Ford 20% allocation—USD 60,000
- Citi 30% allocation—USD 90,000
- McDonalds 20%—USD 60,000

- Installing packages

```
import numpy as np

import pandas as pd

from pandas_datareader  import data as web

import pandas_datareader

import datetime

import matplotlib.pyplot as plt

% matplotlib inline
```

- Retrieving the information concerning the securities

```
start = datetime.datetime(2019, 1, 2 )
end = datetime.datetime(2020, 4, 1 )

Amazon=web.DataReader('AMZN', 'yahoo' , start, end)
Ford=web.DataReader('F', 'yahoo' , start, end)
Citi=web.DataReader('C', 'yahoo' , start, end)
McDonalds=web.DataReader('MCD', 'yahoo' , start, end)
```

- Calculating the returns for the portfolio using closing price

```
for securities in (Amazon,Ford,Citi,McDonalds):
    securities['Return' ]=securities['Close' ] /securities.iloc[0 ][ 'Close' ]
```

- Calculating the returns for the portfolio using closing price

```
for securities , allocation in zip((Amazon,Ford,Citi,MacDonalds),[.3,.2,.3,.2]):
    securities['Allocation' ]=securities['Return' ]*allocation.
```

For the first time the function zip()[5] is used in the present book. The reason for this is that it allows an iteration between the variables. In this case the allocation and the return.

[5] For more information visit: https://realpython.com/python-zip-function/

- Establishing the investment based on the investment of USD 300,000

```
for securities in (Amazon,Ford,Citi,MacDonalds):
    securities['Investment' ]=securities['Allocation' ]* 300000

securities.tail()
```

	High	*Low*	*Open*	*Close*	*Volume*	*Adj Close*	*Return*	*Allocation*	*Investment*
Date									
2020–03-26	170.929993	161.000000	163.990005	167.350006	8,259,900.0	167.350006	0.950528	0.190106	57,031.696612
2020–03-27	169.740005	159.220001	162.779999	164.009995	6,441,400.0	164.009995	0.931557	0.186311	55,893.444319
2020–03-30	170.309998	163.570007	164.919998	168.130005	5,621,700.0	168.130005	0.954959	0.190992	57,297.514670
2020–03-31	169.509995	165.000000	166.839996	165.350006	4,519,900.0	165.350006	0.939169	0.187834	56,350.110779
2020–04-01	161.440002	156.350006	160.220001	158.169998	4,668,900.0	158.169998	0.898387	0.179677	53,903.214937

- Investment per security chosen

```
all_investments =
[Amazon['Investment'],Ford['Investment'],Citi['Investment'],MacDonalds['Investment']]
value_of_portfolio=pd.concat(all_investments,axis=1)
value_of_portfolio.columns=[
'Amazon Investment','Ford Investment','Citi Investment','McDonalds Investment' ]
value_of_portfolio.head()
```

- Analyzing the positions

```
value_of_portfolio['Total Investment']=value_of_portfolio.sum(axis=1)
value_of_portfolio.tail()
```

	Amazon Investment	*Ford Investment*	*Citi Investment*	*McDonalds Investment*	*Total Investment*
Date					
2020–03-26	76,230.987013	39,873.417240	103,164.583988	57,031.696612	276,300.684853
2020–03-27	74,071.714653	39,417.721478	98,187.932530	55,893.444319	267,570.812979
2020–03-30	76,560.782193	38,202.532778	98,815.623769	57,297.514670	270,876.453410
2020–03-31	76,006.053986	36,683.543282	94,421.819299	56,350.110779	263,461.527346
2020–04-01	74,367.984970	33,417.721840	86,329.159424	53,903.214937	248,018.081171

- Plotting the value of the portfolio

```
value_of_portfolio['Total Investment' ].plot(figsize=(10,8 )).
_=plt.xlabel('Investment Performance' )
_=plt.ylabel('Date' )
_=plt.title('Portfolio investment' )
plt.legend();
```

As it can observed in Fig. 23 in Chapter 6, the portfolio began with the investment of USD 300,000 and reached its highest value at USD 369,373 and its lowest value at USD 247,946.82. To calculate this values, it can be done with the min() and max() values (Fig. 5).

```
value_of_portfolio['Total Investment'].max()
396373.7772855786

value_of_portfolio['Total Investment'].min()
247946.82954672351
```

In [118]:

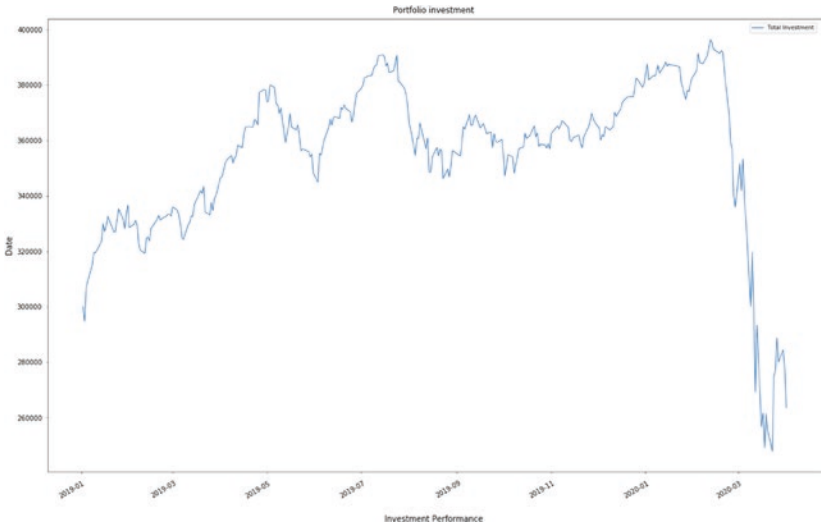


Fig. 5 Total position of the portfolio (*Source* Elaborated by the author with information from Yahoo Finance)

- Plotting the behavior of each security

```
value_of_portfolio.drop('Total Investment',axis=1).plot(figsize=(10,8));
```

In Fig. 24 in Chapter 6, it can be observed that the only security that is gaining is Amazon, McDonald's, Citi and Ford have had a fall losing value in the portfolio (Fig. 6).

- Calculating the Daily Returns

```
value_of_portfolio['Daily Returns'] = value_of_portfolio['Total Investment'].
pct_change(1)
```

- Calculating the mean of the Daily Returns

```
value_of_portfolio['Daily Returns'].mean()
```

```
0.0003811570091341308
```

- Calculating the standard deviation of the Daily Returns

```
value_of_portfolio['Daily Returns'].std()
```

```
0.02105377310080987
```

Fig. 6 Position behavior on each security (*Source* Elaborated by the author with information from Yahoo Finance)

- Calculating the cumulative returns of the portfolio

```
cumulative_return=100 * (value_of_portfolio['Total Investment' ][-1]/value_
of_portfolio['Total Investment' ][0]-1)
12.126738733409015
```

The loss of the portfolio is of negative 12%. This can be analyzed based on the cumulative returns since the beginning of the portfolio.

- Value of the portfolio to date

```
value_of_portfolio['Total Investment' ][-1]
263619.78379977297
```

The portfolio of USD 300,000 has fallen to USD 263,619.78 which is equivalent to a 12.12% fall based on the Covid-19 crisis.

References

Bryant, Bradley James. 2020. *How to Calculate Portfolio Value*. n.d. Accessed January 3, 2020. https://www.sapling.com/5872650/calculate-portfolio-value.

Chen, James. 2019. *Jensen's Measure*. 21 November. Accessed December 20, 2019. https://www.investopedia.com/terms/j/jensensmeasure.asp.

Chen, James. 2019. *Financial Risk*. 15 June. Accessed August 20, 2019. https://www.investopedia.com/terms/f/financialrisk.asp.

David W. Mullins, Jr. 1982. *Does the Capital Asset Pricing Model Work?* n.d. January. Accessed August 13, 2019. https://hbr.org/1982/01/does-the-capital-asset-pricing-model-work.

Fontinelle, Amy. 2019. *Systematic Risk*. 30 September. Accessed October 1, 2019. https://www.investopedia.com/terms/s/systematicrisk.asp.

Hargrave, Marshall. 2019. *Sharpe Ratio*. 17 March. Accessed April 1, 2019. https://www.investopedia.com/terms/s/sharperatio.asp.

Keaton, Will. 2020. *Treynor Ratio*. 22 March. Accessed Apri 1, 2020. https://www.investopedia.com/terms/t/treynorratio.asp.

Mitra, Gautam, and Leela Mitra. 2011. *The Handbook of News Analytics in Finance*. London: Wiley.

Murphy, Chris. 2019. *Information Ratio – IR*. 10 January. Accessed February 20, 2019. https://www.investopedia.com/terms/i/informationratio.asp.

Value at Risk

Abstract Focusing on the creation of portfolios for investment, this chapter aims to understand the risks of the portfolio through methods such as the Value at Risk (VaR) to determine the possible loss or gain of a portfolio. This chapter is based on an investor view and the process for executing decisions that create profitable portfolios in the short and long run.

Keywords Risk · Portfolios · VaR · Backtesting

The concept of Value at Risk (VaR) is one of the most interesting in finance because it analyzes the maximum loss that a portfolio may have (Damodaran 2018). This is another measure of risk that deserves to be separated from portfolio and risk because of the difference that it has with the ratios (Sharpe, Traynor, Information and Jensen) in the previous chapter. To summarize the VaR, it gives the worst loss on a certain time horizon based on the confidence level assigned to the model.

Historical VaR(95)

Since the VaR is based on the confidence level, it may have different results based on a 65%, 90%, 95% or any other confidence interval. The following example is Historical VaR(95), meaning that the confidence interval will be at a 95%.

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9_8

- *Install packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Choose the portfolio*

```
start = datetime.datetime(2018, 1, 2)
end = datetime.datetime(2019, 4, 1)


tickers = ['AAPL','WMT', 'TM','KO','BA']

stocks = pd.DataFrame()
for x in tickers:
    stocks[x] = web.DataReader(x, 'yahoo', start, end)['Close']


stocks.tail()
```

- *Calculate the returns*

```
stocks_return = (stocks/stocks.shift(1))-1
stocks_return.tail()
```

- Assign random portfolio weights that sum to one (1)
- portfolio_weights = np.array(np.random.random(5))

```
portfolio_weights
portfolio_weights = portfolio_weights/np.sum(portfolio_weights)
portfolio_weights
```

This step is interesting because, in the Portfolio and Risk chapter, the purpose was to assign the same return to each of the stocks. In this case the *np.random.random* creates weights for the five (5) stocks but it often gives a number less or higher than 100%. Therefore it has to be balanced by dividing the weights in the sum to obtain a portfolio that sums 100%.

- *Multiply the portfolio with the stocks*

```
weighted_returns_portfolio = stocks_return.mul(portfolio_weights, axis = 1)
```

- *Convert returns to percentages and drop the missing values*

```
stocks_return['Portfolio'] = weighted_returns_portfolio.sum(axis=1).dropna()
stocks_return['Portfolio'] = stocks_return['Portfolio'] * 100
```

- *Calculate the VaR95*

```
var95 = np.percentile(stocks_return['Portfolio'], 5) * 100
var95
```

−1.6739577270187669

Based on the historical returns of the portfolio at a 95% confidence interval, the worst loss is a 1.67% loss, therefore the result is negative.

Historical VaR(99)

For computing the Historical VaR at a 99% confidence level the only change that has to be done is in the last part of the script, changing the *np.percentile* to 1, which means the 1%.

```
var99 = np.percentile(stocks_return['Portfolio'], 5) * 100
var99
```

-2.5793928700853099

At a 99% confidence level the worst loss is 2.58% with the portfolio. Clearly the VaR is higher given that the confidence level is lower. This is rational and therefore it helps understand the process by which the VaR works, given that a higher confidence level will give a higher percentage of loss and a lower confidence level will give a lower percentage of loss.

VaR for the Next 10 Days

One of the most important aspects of calculating a VaR is to calculate the effect on the investment in terms of money. As far, the VaR model has centered on the percentage loss, but for the next example the process is to analyze the VaR if USD 1 million is invested. For example, the same data set will be used.

- *Install packages*

```
#installing packages
```

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Choose the portfolio*

```
start = datetime.datetime(2018, 1, 2)
end = datetime.datetime(2019, 4, 1)


tickers = ['AAPL','WMT', 'TM','KO','BA']
stocks = pd.DataFrame()
for x in tickers:
    stocks[x] = web.DataReader(x, 'yahoo', start, end)['Close']


stocks.tail()
```

- *Calculate the returns*

```
stocks_return = (stocks/stocks.shift(1))-1
stocks_return.tail()
```

- *Assign random portfolio weights that sum to one (1)*

```
portfolio_weights = np.array(np.random.random(5))
portfolio_weights
portfolio_weights = portfolio_weights/np.sum(portfolio_weights)
portfolio_weights
```

- *Multiply the portfolio with the stocks*

```
weighted_returns_portfolio = stocks_return.mul(portfolio_weights,
axis = 1)
```

- *Calculate the returns based on the weights*

```
stocks_return['Portfolio'] = weighted_returns_portfolio.sum(axis=1).dropna()
```

- *Determine the average (mu) of the returns*

```
mu = stocks_return['Portfolio'].mean()
```

- *Determine the standard deviation (sigma) of the returns*

```
sigma = stocks_return['Portfolio'].std()
```

- *Assign a confidence level to the VaR (99% for this example)*

```
confidence = 0.99
```

- *Calculate the alpha*

```
alpha = norm.ppf(1-confidence)
```

For this example, the *norm.ppf* is being used, the reason for this is that it determines the probability density function of one (1) minus the confidence interval. This is useful because it determines the probability of the VaR. It is a similar process to the *np.percentile*.

- *Create a position*

```
position = 1e6
```

The position is the investment on the portfolio. Since the portfolio was created, in this case the investment is USD 1 million. The interesting aspect of using 1e6 for a million is to include a complex number structure that is easier to write. The other choice would have been to write the 1,000,000.

- *Calculate the VaR*

Equation 1: Value at Risk - position

$$VaR = position * (\mu - \sigma * \alpha)$$

μ = mean of the returns of the portfolio
σ = standard deviation of the returns of the portfolio
α = Probability density function of the 1%

```
var = position*(mu-sigma*alpha)
var
```

27088.745452792264

If the investment in the portfolio was of USD 1,000,000, the worst loss at a 99% confidence interval can be of USD 27,088.75. The next step is to obtain the VaR for the next 10 days, trying to identify what will be the loss of the portfolio.

- *Create a variable for 10 days*

```
days = 10
```

- *Determine the worst loss for the next 10 days*

$$VaR\ for\ 10\ days = position * (\mu * days - \sigma * \alpha * \sqrt{days})$$

```
var_10_days = position *(mu*days-sigma*alpha*np.sqrt(days))
var_10_days
```

88644.949585607217

The worst loss for the next 10 days based on the portfolio that depends on the stocks that have been chosen and the weights of the stocks, could be of USD 88,644.95 or approximately 88.64% of the total investment. Consider that this effect is at a 99% confidence interval. If the example had been done with a 95% of confidence interval, the result would have been as follows:

- *Assign a confidence interval of 95%*

```
confidence = 0.95
```

- *Obtain the alpha*

```
alpha = norm.ppf(1-confidence)
```

- *Determine the worst loss for the next 10 days*

```
var_10_days = position *(mu*days-sigma*alpha*np.sqrt(days))
var_10_days
```

63954.684614643818

The result is a worst loss much smaller than the one determined at a 99% confidence level. In this example the loss is approximately 63.95% of the total investment. The result also varies if the days are reduced, for example to 5 days.

- *Worst loss for the next 5 days at a 95% confidence level*

```
days_2 = 5

var_5_days = position *(mu*days_2 - sigma*alpha*np.sqrt(days))
var_5_days
```

61773.537991536054

Historical Drawdown

A historical drawdown is often included with the VaR because it analyzes the decline in the specific period that the portfolio is being analyzed and based on the cumulative growth analyzes the peak and therefore the fall or drawdown of the portfolio (Mitchell 2019). The process is similar to the VaR but it uses the negative returns of the portfolio.

- *Installing packages*

```
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

- *Choose the portfolio*

```
start = datetime.datetime(2018, 1, 2)
end = datetime.datetime(2019, 4, 1)


tickers = ['AAPL','WMT', 'TM','KO','BA']
stocks = pd.DataFrame()
for x in tickers:
   stocks[x] = web.DataReader(x, 'yahoo', start, end)['Close']


stocks.tail()
```

- *Calculate the returns*

```
stocks_return = (stocks/stocks.shift(1))-1
stocks_return.tail()
```

- *Assign random portfolio weights that sum to one (1)*

```
portfolio_weights= np.array(np.random.random(5))
portfolio_weights
portfolio_weights = portfolio_weights/np.sum(portfolio_weights)
portfolio_weights
```

- *Multiply the portfolio with the stocks*

```
weighted_returns_portfolio = stocks_return.mul(portfolio_weights,
axis = 1)
```

- *Calculate the returns based on the weights*

```
stocks_return['Portfolio'] = weighted_returns_portfolio.sum(axis=1).dropna()
```

- *Calculate the cumulative returns*

```
CumulativeReturns = ((1+stocks_return["Portfolio"]).cumprod()-1)
```

- *Plot the cumulative returns* (Fig. 1)

```
CumulativeReturns.plot()
_ = plt.xlabel('Dates')s


_ = plt.ylabel('Returns')


_ = plt.title('Cumulative Returns - Portfolio')


plt.show()
```

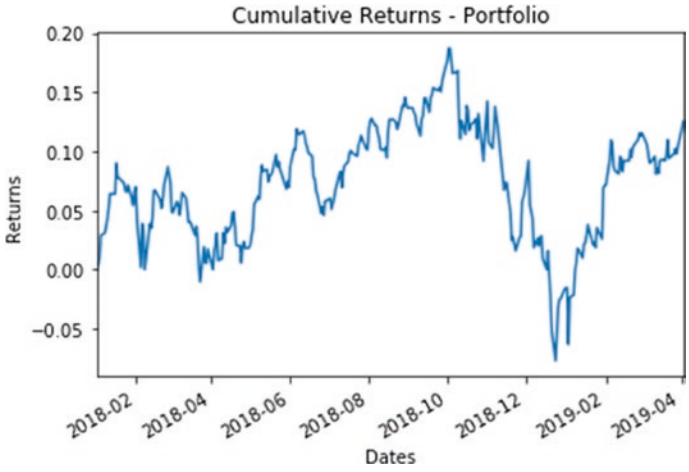


Fig. 1 Cumulative Return of the portfolio (*Source* Elaborated by the author with information from Yahoo Finance)

- *Determine the running maximum*

```
running_maximum = np.maximum.accumulate(CumulativeReturns)
running_maximum.tail()
```

The running maximum is taking the maximum of the cumulative returns and accumulates it so that there is a constant return. This is because the drawdown formula divides the cumulative return of the portfolio into the running maximum. The *np.maximum.accumulate* accumulates all the elements.

- *Establish that the running maximum should not go below zero*

```
running_maximum = running_maximum < 1
```

- *Calculate the drawdown*

```
portfolio_drawdown = (CumulativeReturns)/running_max − 1
```

- *Plot the drawdown* (Fig. 2)

```
drawdown.plot()


_ = plt.xlabel('Dates')


_ = plt.ylabel('Returns')


_ = plt.title('Drowdown' - Portfolio')


plt.show()
```

As can be seen in the drawdown the returns are negative and it demonstrates that the worst loss is between the end of 2018 and February 2019. It is an interesting approach based on the drawdown to observe how the portfolio could behave to a VaR. In this case the worst loss is 1.4% being a very stable portfolio.

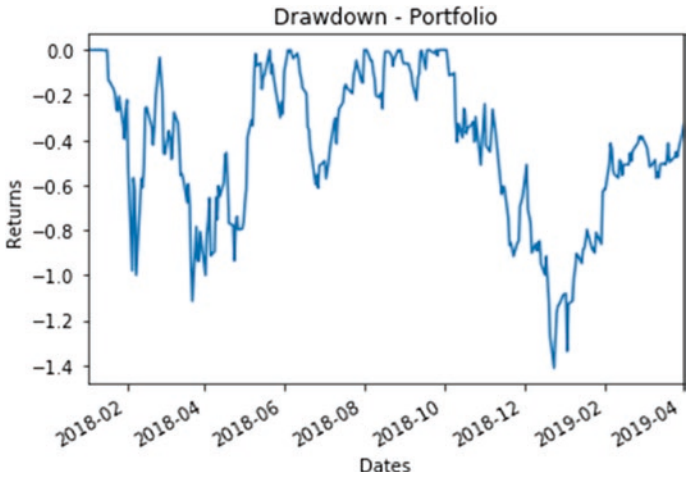


Fig. 2 Drawdown of the portfolio (*Source* Elaborated by the author with information from Yahoo Finance)

Wrapping Up the Book—Understanding Performance

The book has centered itself on stock trading, and the purpose of this chapter is to understand performance by using the *f.fn()* package for creating a report. I believe this is one of the most interesting tools that can be used for a quick decision making on investment that does not involve a graphical decision. This can be applied to a portfolio and individual stocks. The application of the portfolio is as follows:

Portfolio Performance using f.fn()

– Import libraries

```
import ffn
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

– Select stocks

```
start = datetime.datetime(2019, 1, 1)
end = datetime.datetime(2020, 12, 30)

tickers = ['ZM','AMZN', 'DOCU','PTON']
stocks = pd.DataFrame()
for x in tickers:
    stocks[x] = web.DataReader(x, 'yahoo', start, end)['Close']
```

– Calculate returns using *f.fn ()*

```
stocks_return = stocks.to_returns().dropna()
stocks_return.tail()
```

– Calculating mean variance returns with *f.fn ( )*

```
mean_variance_weights =
stocks_return.calc_mean_var_weights().as_format('.2%')
mean_variance_weights
```

```
ZM      16.84%
AMZN    22.13%
DOCU    25.83%
PTON    35.21%
dtype: object
```

– Applying the mean variance returns to portfolio weights,

```
portfolio_weights = (0.1759,0.2111, 0.2592, 0.3538, 0.00)
portfolio_weights
```

– Creating the portfolio based on the returns

```
portfolio_weights = portfolio_weights/np.sum(portfolio_weights)
weighted_returns_portfolio = stocks_return.mul(portfolio_weights, axis =
1)
stocks_return['Portfolio'] =
weighted_returns_portfolio.sum(axis=1).dropna()
stocks_return.tail()
```

	ZM	*AMZN*	*DOCU*	*PTON*	*BA*	*Portfolio*
Date						
2020-12-23	−0.061418	−0.006627	−0.030008	0.009615	0.004159	−0.016579
2020-12-24	−0.022689	−0.003949	0.003606	−0.000246	−0.011562	−0.003977
2020-12-28	−0.063385	0.035071	−0.064222	−0.064774	−0.004881	−0.043309
2020-12-29	0.006716	0.011584	−0.004537	−0.013668	0.000740	−0.002385
2020-12-30	−0.000989	−0.010882	−0.009905	0.032378	0.001942	0.006417

– Calculate performance

```
performance = stocks_return.calc_stats()
```

– Display performance

```
performance.display()
```

Stat	*ZM*	*AMZN*	*DOCU*	*PTON*	*Portfol*
Start	2019-09-27	2019-09-27	2019-09-27	2019-09-27	2019-09-27
End	2020-12-30	2020-12-30	2020-12-30	2020-12-30	2020-12-30
Risk-free rate	0.00%	0.00%	0.00%	0.00%	0.00%
Total Return	−97.87%	31.57%	−64.69%	−260.40%	−126.09%
Daily Sharpe	−1.50	−1.50	0.07	-	−0.45
Daily Sortino	−1.51	−1.55	0.28	inf	−1.07
CAGR	−95.30%	24.34%	−56.24%	-	-
Max Drawdown	−1328.75%	−313.08%	−200.92%	−398.81%	−899.28
Calmar Ratio	−0.07	0.08	−0.28	-	-
MTD	−106.91%	27.37%	−322.50%	−52.12%	−75.40%
3 m	−109.99%	−988.87%	−191.30%	−364.48%	1356.52%
6 m	−104.94%	−137.18%	−414.89%	230.78%	−55.58%
YTD	−105.29%	−2215.67%	−302.92%	26.28%	−53.76%
1Y	−143.96%	−11.19%	15.13%	26.70%	41.04%
3Y (ann.)	−95.30%	24.34%	−56.24%	-	-
5Y (ann.)	-	-	-	-	-
10Y (ann.)	-	-	-	-	-
Since Incep. (ann.)	−95.30%	24.34%	−56.24%	-	-
Daily Sharpe	−1.50	−1.50	0.07	-	−0.45
Daily Sortino	−1.51	−1.55	0.28	inf	−1.07
Daily Mean (ann.)	−114,734.65%	−76,805.36%	1704.15%	inf%	−11,126.29%
Daily Vol (ann.)	76,370.55%	51,048.23%	22,749.18%	-	24,907.54%
Daily Skew	−16.67	−13.99	13.26	-	8.40
Daily Kurt	288.21	224.97	213.60	-	108.93
Best Day	3995.93%	8524.06%	23,064.40%	inf%	20,939.52%
Worst Day	−83,830.58%	−52,654.19%	−3831.82%	−5624.22%	−5896.52%
Monthly Sharpe	−1.32	0.66	−0.98	−0.50	−1.31
Monthly Sortino	−1.79	2.25	−1.53	−0.62	−1.31
Monthly Mean (ann.)	−1280.15%	9042.18%	−1050.99%	−1291.43%	−14,979.80%
Monthly Vol (ann.)	972.34%	13,779.63%	1077.20%	2606.03%	11,440.89%
Monthly Skew	−0.18	3.02	0.45	−1.56	−3.23
Monthly Kurt	0.81	11.13	1.07	7.23	10.79

Stat	*ZM*	*AMZN*	*DOCU*	*PTON*	*Portfol*
Best Month	451.85%	14,246.47%	615.58%	1409.53%	113.76%
Worst Month	−693.26%	−4548.17%	−591.84%	−2380.16%	−12,427.53%
Yearly Sharpe	-	-	-	-	-
Yearly Sortino	-	-	-	-	-
Yearly Mean	−105.29%	−2215.67%	−302.92%	26.28%	−53.76%
Yearly Vol	-	-	-	-	-
Yearly Skew	-	-	-	-	-
Yearly Kurt	-	-	-	-	-
Best Year	−105.29%	−2215.67%	−302.92%	26.28%	−53.76%
Worst Year	−105.29%	−2215.67%	−302.92%	26.28%	−53.76%
Avg. Drawdown	−345.25%	−201.60%	−167.47%	−190.98%	−267.83%
Avg. Drawdown Days	64.29	63.57	63.71	55.75	49.11
Avg. Up Month	213.22%	2862.18%	285.41%	690.44%	52.58%
Avg. Down Month	−223.00%	−652.26%	−223.22%	−307.13%	−1573.54%
Win Year %	0.00%	0.00%	0.00%	1000.00%	0.00%
Win 12 m %	20.00%	40.00%	60.00%	40.00%	40.00%

The result of the portfolio, during the uncertainty of 2020 with a deplorable negative 126.09% of loss, a drawdown of negative 267.83% with an average of 49.11 days. The best month of the portfolio created a 113.76% but the implication of a worse month surpasses that. With this information corrections and backtesting can be created for better performance when analyzing the data.

If the data wants to be seen with only one stock, the process is similar as when handling the DataFrame.

```
performance = stocks_return.calc_stats()
```

```
start                    2019-09-27 00:00:00
end                      2020-12-30 00:00:00
rf                       0
total_return             −1.26094
cagr                     NaN
max_drawdown             −8.99284
calmar                   NaN
mtd                      −0.754048
three_month              13.5652
six_month                −0.555818
ytd                      −0.53764
one_year                 0.410417
three_year               NaN
five_year                NaN
ten_year                 NaN
incep                    NaN
daily_sharpe             −0.446704
daily_sortino            −1.06647
```

daily_mean	−111.263
daily_vol	249.075
daily_skew	8.39942
daily_kurt	108.93
best_day	209.395
worst_day	−58.9652
monthly_sharpe	−1.30932
monthly_sortino	−1.31108
monthly_mean	−149.798
monthly_vol	114.409
monthly_skew	−3.23038
monthly_kurt	10.7853
best_month	1.1376
worst_month	−124.275
yearly_sharpe	NaN
yearly_sortino	NaN
yearly_mean	−0.53764
yearly_vol	NaN
yearly_skew	NaN
yearly_kurt	NaN
best_year	−0.53764
worst_year	−0.53764
avg_drawdown	−2.67834
avg_drawdown_days	49.1111
avg_up_month	0.525848
avg_down_month	−15.7354
win_year_perc	0
twelve_month_win_perc	0.4

Fund Performance using f.fn()

As seen before, there are other analyses using performance that can be done, when analyzing the prices of a stock or in this case, a fund. Although funds have not been discussed in the book, it is important to understand that the statistical methods are similar, the interpretation changes regarding the instrument. The process is as follow:

– Importing libraries

```
import ffn
import numpy as np
import pandas as pd
from pandas_datareader import data as web
import pandas_datareader
import datetime
import matplotlib.pyplot as plt
%matplotlib inline
```

– Selecting funds using f.fn()

```
funds = ffn.get('MSAUX:Close,MIOPX:Close, MGGPX:Close, MFAPX:Close,
SPY:Close', start='2018-01-01', end='2021-01-30')
funds.tail( )
```

	msauxclose	*miopxclose*	*mggpxclose*	*mfapxclose*	*spyclose*
Date					
2021-01-25	34.349998	42.810001	43.990002	26.910000	384.390015
2021-01-26	33.910000	42.509998	43.630001	26.969999	383.790009
2021-01-27	33.110001	41.509998	42.290001	26.430000	374.410004
2021-01-28	33.150002	41.810001	43.060001	26.610001	377.630005
2021-01-29	32.919998	41.250000	42.509998	26.170000	370.070007

– Creating an SMA for the Morgan Stanley Institutional Fund, Inc. Asia Opportunity Portfolio Class A (MSAUX) using *talib* with 20 days

```
funds['MA-msaux'] = talib.SMA(funds['msauxclose'],20)
```

– Plotting the SMA for comparison (Fig. 3)

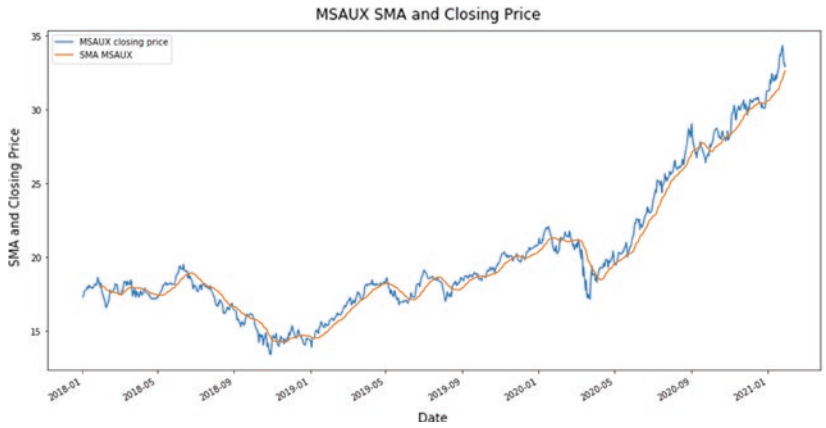


Fig. 3 MSAUX SMA

```
funds['msauxclose'].plot(label='MSAUX closing price',figsize=(16,8))
funds['MA-msaux'].plot(label= 'SMA MSAUX')

_ = plt.xlabel('Date')

_ = plt.ylabel('SMA and Closing Price')

_ = plt.title('MSAUX SMA and Closing Price')

plt.legend();
```

– Creating an EMA for the Morgan Stanley Institutional Fund, Inc. Asia Opportunity Portfolio Class A (MSAUX) using *talib* with 20 days (Fig. 4)

```
funds['EMA-msaux'] = talib.EMA(funds['msauxclose'], timeperiod = 20)

funds['msauxclose'].plot(label='MSAUX closing price',figsize=(16,8))
funds['EMA-msaux'].plot(label= 'EMA MSAUX')

_ = plt.xlabel('Date')

_ = plt.ylabel('EMA and Closing Price')

_ = plt.title('MSAUX EMA and Closing Price')

plt.legend();
```

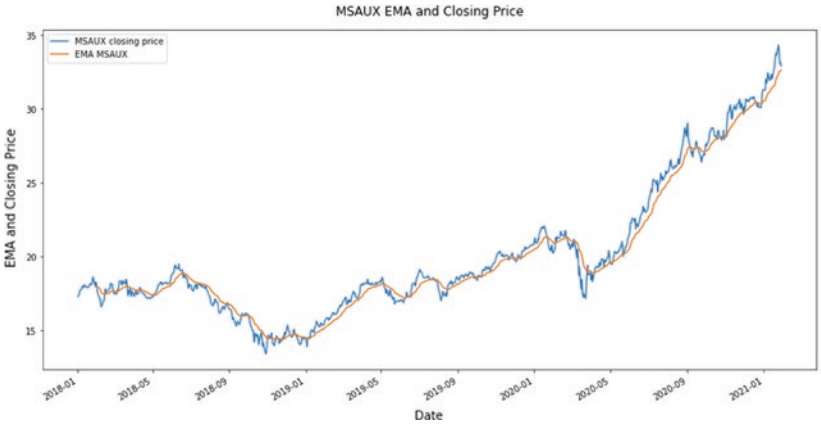


Fig. 4 MSAUX EMA

– Creating Bollinger Bands for the Morgan Stanley Institutional Fund, Inc. Asia Opportunity Portfolio Class A (MSAUX) using *talib* with 20 days (Fig. 5)

```
funds['up_band'], funds['mid_band'], funds['low_band'] =
talib.BBANDS(funds['msauxclose'], timeperiod =20)
funds.tail()

funds['msauxclose'].plot(label='MSAUX closing price',figsize=(16,8))
funds['up_band'].plot(label= 'Upper Band')
funds['mid_band'].plot(label= 'Midle Band')
funds['low_band'].plot(label= 'Lower Band')

_ = plt.xlabel('Date')

_ = plt.ylabel('Bollinger Bands and Closing Price')

_ = plt.title('MSAUX Bollinger Bands and Closing Price')

plt.legend();
```

– Creating RSI for the Morgan Stanley Institutional Fund, Inc. Asia Opportunity Portfolio Class A (MSAUX) using *talib* with 14 days (Fig. 6)

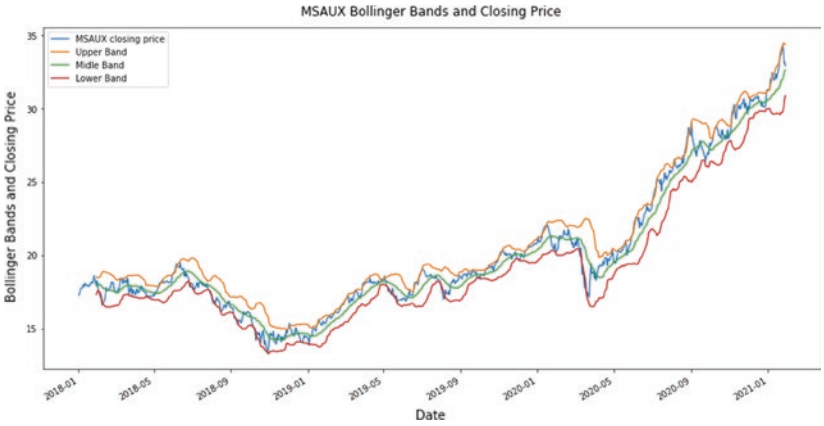


Fig. 5 MSAUX Bollinger Bands

```
funds['RSI'] = talib.RSI(funds['msauxclose'],14)


funds['RSI'].plot(label='MSAUX closing price',figsize=(16,8))


_ = plt.xlabel('Date')

_ = plt.ylabel('MSAUX RSI')

_ = plt.title('MSAUX RSI')

plt.legend();
```

– Calculate logarithmic returns for the funds (excluding the above calculations)

```
returns = funds.to_log_returns().dropna()
returns.head()
```

	msauxclose	*miopxclose*	*mggpxclose*	*mfapxclose*	*spyclose*
Date					
2018-01-03	0.004619	0.004873	0.009265	0.002963	0.006305
2018-01-04	0.008032	0.005289	0.005693	0.006488	0.004206
2018-01-05	0.011929	0.007445	0.009994	0.005277	0.006642
2018-01-08	0.004507	−0.001310	0.002591	−0.002928	0.001827
2018-01-09	0.010067	0.005663	0.004732	0.005265	0.002261

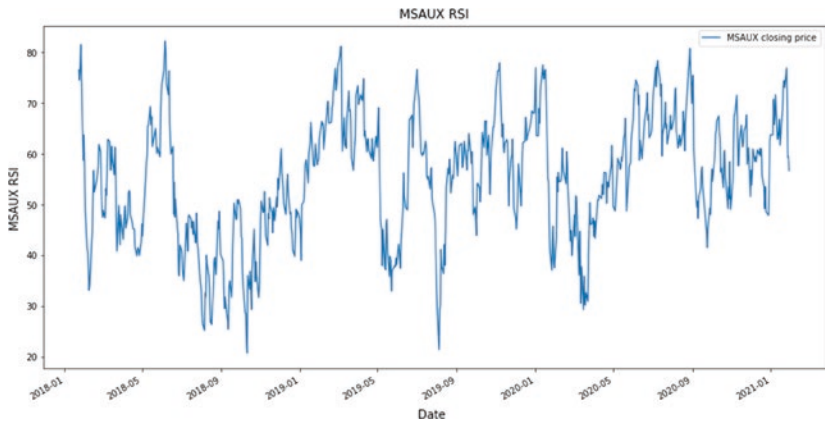


Fig. 6 MSAUX RSI

– Calculate a correlation matrix

```
returns.corr().as_format('.2f')
```

	msauxclose	*miopxclose*	*mggpxclose*	*mfapxclose*	*spyclose*
msauxclose	1.00	0.88	0.80	0.79	0.69
miopxclose	0.88	1.00	0.94	0.95	0.83
mggpxclose	0.80	0.94	1.00	0.90	0.89
mfapxclose	0.79	0.95	0.90	1.00	0.87
spyclose	0.69	0.83	0.89	0.87	1.00

– Rebasing the funds for comparison on one another (Fig. 7)

```
unds.rebase().plot(figsize=(12,5))


_ = plt.xlabel('Date')

_ = plt.ylabel('Closing Price')

_ = plt.title('Rebase of the closing price ')

plt.legend();
```

– Calculate performance using *f.fn()* (Fig. 8)

```
funds = funds.dropna()

performance = funds.calc_stats()

performance.plot(figsize=(12,5))


_ = plt.xlabel('Date')


_ = plt.title('Funds montly Progression')

plt.legend();
```

– Display performance indicators

```
performance.display()
```

Fig. 7 Rebase of the closing price in funds and ETF

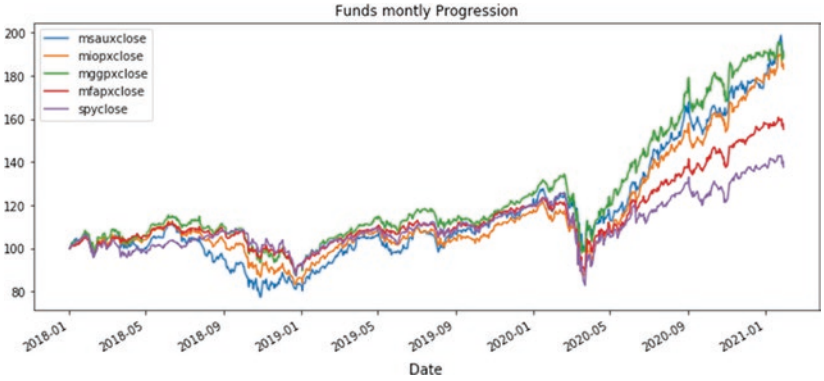


Fig. 8 Funds and ETF montlhy progression

Stat	*msauxclose*	*miopxclose*	*mggpxclose*	*mfapxclose*	*spyclose*
Start	2018-01-02	2018-01-02	2018-01-02	2018-01-02	2018-01-02
End	2021-01-29	2021-01-29	2021-01-29	2021-01-29	2021-01-29
Risk-free rate	0.00%	0.00%	0.00%	0.00%	0.00%
Total Return	90.51%	83.17%	88.43%	55.31%	37.69%
Daily Sharpe	1.05	1.04	1.02	0.85	0.57
Daily Sortino	1.74	1.61	1.58	1.28	0.84
CAGR	23.32%	21.76%	22.88%	15.40%	10.96%
Max Drawdown	−31.35%	−28.34%	−27.19%	−27.78%	−34.10%
Calmar Ratio	0.74	0.77	0.84	0.55	0.32

Stat	*msauxclose*	*miopxclose*	*mggpxclose*	*mfapxclose*	*spyclose*
MTD	5.21%	1.68%	−1.23%	−1.54%	−1.02%
3 m	15.31%	15.77%	9.08%	11.27%	12.15%
6 m	27.60%	26.57%	21.81%	17.72%	13.83%
YTD	5.21%	1.68%	−1.23%	−1.54%	−1.02%
1Y	58.42%	58.96%	47.04%	29.55%	13.30%
3Y (ann.)	21.97%	20.13%	20.80%	14.13%	9.14%
5Y (ann.)	23.32%	21.76%	22.88%	15.40%	10.96%
10Y (ann.)	23.32%	21.76%	22.88%	15.40%	10.96%
Since Incep. (ann.)	23.32%	21.76%	22.88%	15.40%	10.96%
Daily Sharpe	1.05	1.04	1.02	0.85	0.57
Daily Sortino	1.74	1.61	1.58	1.28	0.84
Daily Mean (ann.)	23.51%	21.93%	23.22%	16.13%	13.03%
Daily Vol (ann.)	22.41%	20.98%	22.68%	18.87%	22.79%
Daily Skew	−0.34	−0.76	−0.61	−1.00	−0.70
Daily Kurt	2.10	9.41	8.60	14.99	13.18
Best Day	5.95%	8.13%	7.83%	7.37%	9.06%
Worst Day	−7.92%	−9.93%	−10.55%	−9.81%	−10.94%
Monthly Sharpe	1.06	1.05	1.08	0.95	0.58
Monthly Sortino	2.10	2.14	2.27	1.72	0.98
Monthly Mean (ann.)	21.90%	20.24%	20.82%	14.31%	10.82%
Monthly Vol (ann.)	20.62%	19.35%	19.35%	15.05%	18.72%
Monthly Skew	−0.49	−0.21	−0.04	−0.58	−0.38
Monthly Kurt	−0.29	−0.15	−0.02	0.02	0.61
Best Month	10.70%	11.89%	12.75%	7.81%	12.70%
Worst Month	−13.87%	−11.99%	−11.10%	−9.43%	−13.00%
Yearly Sharpe	1.36	1.13	1.05	1.07	0.98
Yearly Sortino	inf	inf	40.77	22.41	24.89
Yearly Mean	33.30%	30.38%	28.99%	19.96%	14.64%
Yearly Vol	24.53%	26.77%	27.70%	18.65%	14.96%
Yearly Skew	−1.60	−0.72	−0.93	−1.70	−0.45
Yearly Kurt	-	-	-	-	-
Best Year	50.51%	54.67%	53.17%	31.85%	28.79%
Worst Year	5.21%	1.68%	−1.23%	−1.54%	−1.02%
Avg. Drawdown	−3.55%	−3.14%	−3.38%	−2.30%	−2.64%
Avg. Drawdown Days	30.15	26.30	22.62	19.40	22.34
Avg. Up Month	5.79%	5.39%	5.63%	3.81%	3.84%
Avg. Down Month	−4.41%	−3.49%	−3.13%	−3.44%	−4.97%
Win Year %	100.00%	100.00%	66.67%	66.67%	66.67%
Win 12 m %	80.77%	76.92%	84.62%	80.77%	88.46%

– Calculate the drawdown series

```
funds.to_drawdown_series().tail()
```

	msauxclose	*miopxclose*	*mggpxclose*	*mfapxclose*	*spyclose*
Date					
2021-01-25	0.000000	0.000000	−0.003398	−0.006645	0.000000
2021-01-26	−0.012809	−0.007008	−0.011554	−0.004430	−0.001561
2021-01-27	−0.036099	−0.030367	−0.041912	−0.024363	−0.025963
2021-01-28	−0.034934	−0.023359	−0.024468	−0.017719	−0.017586
2021-01-29	−0.041630	−0.036440	−0.036928	−0.033961	−0.037254

The example above that compared funds and an ETF, compared in performance the MSAUX as the best option based on the total return, the CAGR and the 12-month percentage. The data also shows that it is the riskier asset in its class with an average drawdown of −3.55%, the highest average drawdown days with 30.15 and a high average drawdown per month. The information of the performance report can be applied to everything learned in this book.

Works Cited

Damodaran, Aswath. 2018. Value at Risk (VAR). *New York University*. n.d. Accessed February 20, 2019. http://people.stern.nyu.edu/adamodar/pdfiles/papers/VAR.pdf.

Mitchell, Cory. 2019. *Drawdown definition and example*. 25 June. Accessed July 30, 2019. https://www.investopedia.com/terms/d/drawdown.asp.

Works Cited

365 Data Science. 2020. *Why Python for data science and why Jupyter to code in Python Articles 11 min read*. n.d. Accessed March 2, 2020. https://365datascience.com/why-python-for-data-science-and-why-jupyter-to-code-in-python/.

Anaconda. 2020. *Anaconda distribution*. Accessed March 2, 2020. https://www.anaconda.com/distribution/.

Bang, Julie. 2019. *Candlestick bar*. Investopedia.

Basurto, Stefano. 2020. *Python trading toolbox: Introducing OHLC charts with Matplotlib*. 07 January. Accessed March 31, 2020. https://towardsdatascience.com/trading-toolbox-03-ohlc-charts-95b48bb9d748.

Bloomberg Corporation. 2019. *Bloomberg puts the power of Python* XE "Python" *in hedgers' hands*. 8 March. Accessed March 2, 2020. https://www.bloomberg.com/professional/blog/bloomberg-puts-power-python-hedgers-hands/.

Bollinger, John. 2018. *John Bollinger answers "What are Bollinger Bands?"*. n.d. Accessed March 2, 2020. https://www.bollingerbands.com/bollinger-bands.

Bolsa de Madrid. 2020. *Electronic Spanish Stock Market Interconnection System (SIBE)*. n.d. Accessed March 23, 2020. http://www.bolsamadrid.es/ing/Inversores/Agenda/HorarioMercado.aspx.

Brooks, Chris. 2008. *Introductory econometrics for finance*. Boston: Cambridge University Press.

Bryant, Bradley James. 2020. *How to calculate portfolio value*. n.d. Accessed January 3, 2020. https://www.sapling.com/5872650/calculate-portfolio-value.

Burgess, Matthew, and Sarah Wells. 2020. *Giant wealth fund seeks managers who can beat frothy market*. 9 February. Accessed March 2, 2020. https://finance.yahoo.com/news/giant-wealth-fund-seeks-managers-230000386.html.

Chen, James. 2019a. *Financial risk*. 15 June. Accessed August 20, 2019. https://www.investopedia.com/terms/f/financialrisk.asp.

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9

Chen, James. 2019b. *Jensen's Measure*. 21 November. Accessed December 20, 2019. https://www.investopedia.com/terms/j/jensensmeasure.asp.

Chen, James. 2019c. *Line Chart*. 12 August. Accessed March 25, 2020. https://www.investopedia.com/terms/l/linechart.asp.

Damodaran, Aswath. 2018. Value at Risk (VAR). *New York University*. n.d. Accessed February 20, 2019. http://people.stern.nyu.edu/adamodar/pdfiles/papers/VAR.pdf.

Das, Sejuti. 2020. *Analytics India Magazine*. 27 02. Accessed March 17, 2020. https://analyticsindiamag.com/why-jupyter-notebooks-are-so-popular-among-data-scientists/.

Mullins, David W. 1982. *Does the capital asset pricing model work?* n.d January. Accessed August 13, 2019. https://hbr.org/1982/01/does-the-capital-asset-pricing-model-work.

Fontinelle, Amy. 2019. *Systematic risk*. 30 September. Accessed October 1, 2019. https://www.investopedia.com/terms/s/systematicrisk.asp.

Ganti, Akhilesh. 2019. *Central Limit Theorem (CLT)*. 13 September. Accessed April 2, 2019. https://www.investopedia.com/terms/c/central_limit_theorem.asp.

Halton, Clay. 2019. *Line graph*. 21 August. Accessed March 25, 2020. https://www.investopedia.com/terms/l/line-graph.asp.

Hargrave, Marshall. 2019. *Sharpe ratio*. 17 March. Accessed April 1, 2019. https://www.investopedia.com/terms/s/sharperatio.asp.

Hargrave, Marshall. 2020. *Standard deviation definition*. 1 February. Accessed February 20, 2020. https://www.investopedia.com/terms/s/standarddeviation.asp.

Hayes, Adam. 2018. *Volume definition*. 4 February. Accessed March 25, 2020. https://www.investopedia.com/terms/v/volume.asp.

Hayes, Adam. 2019a. *Correlation definition*. 20 June. Accessed January 1, 2020. https://www.investopedia.com/terms/c/correlation.asp.

Hayes, Adam. 2019b. *Correlation definition*. 20 June. Accessed October 8, 2019. https://www.investopedia.com/terms/c/correlation.asp.

Hayes, Adam. 2019c. *Variance*. 2 September. Accessed August 3, 2019. https://www.investopedia.com/terms/v/variance.asp.

Hayes, Adam. 2020a. *Exponential Moving Average - EMA definition*. 8 July. Accessed April 2, 2020. https://www.investopedia.com/terms/e/ema.asp.

Hayes, Adam. 2020b. *Moving Average (MA)*. 31 March. Accessed March 31, 2020. https://www.investopedia.com/terms/m/movingaverage.asp.

Hunner, Trey. 2018. *Trey Hunner*. 11 October. Accessed March 23, 2020. https://treyhunner.com/2018/10/asterisks-in-python-what-they-are-and-how-to-use-them/.

Jain, Diva. 2018. *Skew and Kurtosis: 2 Important statistics terms you need to know in Data Science*. 23 August. Accessed August 12, 2019. https://codeburst.io/2-important-statistics-terms-you-need-to-know-in-data-science-skewness-and-kurtosis-388fef94eeaa.

Kalla, Siddharth. 2020. *Range (Statistics)*. n.d. Accessed January 4, 2020. https://explorable.com/range-in-statistics.

Kan, Chi Nok. 2018. *Data Science 101: Is Python better than R?* 1 August. Accessed March 2, 2020. https://towardsdatascience.com/data-science-101-is-python-better-than-r-b8f258f57b0f.

Keaton, Will. 2019. *Quantitative Analysis (QA)*. 18 April. Accessed January 15, 2020. https://www.investopedia.com/terms/q/quantitativeanalysis.asp.

Keaton, Will. 2020. *Treynor ratio*. 22 March. Accessed Apri 1, 2020. https://www.investopedia.com/terms/t/treynorratio.asp.

Kenton, Will. 2019. *Kurtosis*. 17 February. Accessed July 30, 2019. https://www.investopedia.com/terms/k/kurtosis.asp.

Kevin, S. 2015. *Security analysis and portfolio management*. Delhi: PHI.

London Stock Exchange Group. 2020. *London Stock Exchange Group Business Day*. n.d. Accessed March 25, 2020. https://www.lseg.com/areas-expertise/our-markets/london-stock-exchange/equities-markets/trading-services/business-days.

Mastromatteo, Davide. 2020. *Python args and kwargs: Demystified*. 09 September. Accessed March 31, 2020. https://realpython.com/python-kwargs-and-args/.

Milton, Adam. 2020. *Simple, exponential, and weighted moving averages*. 09 November. Accessed March 31, 2020. https://www.thebalance.com/simple-exponential-and-weighted-moving-averages-1031196.

Mitchell, Cory. 2019a. *Don't trade based on MACD divergence until you read this*. 19 November. Accessed April 1, 2020. https://www.thebalance.com/dont-trade-based-on-macd-divergence-until-you-read-this-1031217.

Mitchell, Cory. 2019b. *Drawdown definition and example*. 25 June. Accessed July 30, 2019. https://www.investopedia.com/terms/d/drawdown.asp.

Mitchell, Cory. 2019. *Understanding basic candlestick charts*. 19 December. Accessed March 30, 2020. https://www.investopedia.com/trading/candlestick-charting-what-is-it/.

Mitchell, Cory. 2020. *How to use volume to improve your trading*. 25 February. Accessed March 27, 2020. https://www.investopedia.com/articles/technical/02/010702.asp.

Mitra, Gautam, and Leela Mitra. 2011. *The handbook of news analytics in finance*. United Kingdom: Wiley.

Murphy, Casey. 2020. *Investopedia*. 16 November. Accessed January 01, 2021. https://www.investopedia.com/trading/introduction-to-parabolic-sar/.

Murphy, Chris. 2019. *Information Ratio – IR*. 10 January. Accessed February 20, 2019. https://www.investopedia.com/terms/i/informationratio.asp.

Murphy, John J. 1999. *Technical analysis of the financial markets*. New York: New York Institute of Finance.

NYSE. 2020. *TAQ closing prices*. n.d. Accessed March 25, 2020. https://www.nyse.com/market-data/historical/taq-nyse-closing-prices.

O`Reilly. 2020. *5 key areas for tech leaders to watch in 2020*. 18 February. Accessed March 2, 2020. https://www.oreilly.com/radar/oreilly-2020-platform-analysis/.

Pfeiffer, Frank. 2019. *R versus Python: Which programming language is better for data science projects in Finance?* 28 May. Accessed March 2, 2020. https://finance-blog.arvato.com/r-versus-python-in-finance/.

Posey, Luke. 2019. *Implementing MACD in Python* XE "Python". 30 March. Accessed April 2, 2020. https://towardsdatascience.com/implementing-macd-in-python-cc9b2280126a.

Python Organization. 2020. *Python organization*. 07 January. Accessed March 02, 2020. https://docs.python.org/2/faq/general.html#what-is-python.

Python. 2006. *2.3.4 Numeric types -- int, float, long, complex*. 18 October. Accessed March 23, 2020. https://docs.python.org/2.4/lib/typesnumeric.html.

Python. 2020. *Data structures*. 23 March. Accessed March 23, 2020. https://docs.python.org/3/tutorial/datastructures.html.

Shaik, Naushad. 2018. *5 reasons why learning Python is the best decision*. 21 September. Accessed March 2, 2020. https://medium.com/datadrivenin-vestor/5-reasons-why-i-learned-python-and-why-you-should-learn-it-as-well-917f781aea05.

Trochim, William M.K. 2020. *Correlation*. 10 March. Accessed March 12, 2020. https://conjointly.com/kb/correlation-statistic/.

Wan, Xiang, Wengian Wang, Jiming Liu, and Tiejun Tong. 2014. *Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range*. 19 December. Accessed January 03, 2019. https://bmcmedresmethodol.biomedcentral.com/articles/10.1186/1471-2288-14-135.

Index

M. Garita, *Applied Quantitative Finance*,
https://doi.org/10.1007/978-3-030-29141-9